Centre for Knowledge Analytics and

Ontological Engineering

http://www.kanoe.org

(World Bank/TEQIP II Funded)

MOEKA - Memoranda in Ontological Engineering and Knowledge Analytics

MOEKA.2013.5

SWITH: Semantic Web-based Intelligent Technical Helpdesk

Aparna N., Apoorva Rao B. and Karthik R Prasad

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

Ring Road, Banashankari III Stage, Bangalore-560 085, India

Publication of this Technical Report and the research work presented here was supported in part by the World Bank/Government of India research grant under the TEQIP programme (subcomponent 1.2.1) to the Centre for Knowledge Analytics and Ontological Engineering (KAnOE), http://kanoe.org, at PES University (formerly PES Institute of Technology), Bangalore, India.

Contents

Abstract

Software developers face various issues in the software development lifecycle. The solution, more often than not, is to go through some technical documentation. Unfortunately, documentation can be very unstructured, distributed and difficult to comprehend.

Our solution to this problem involves two phases - Structuring technical documentation by semanticizing it; and thereby facilitating the easy visualization of relevant knowledge through an application.

Structuring the documents - by creating a linked data store - enables the easy retrieval of relevant information. Linked data has the special quality of being highly interlinked and structured. It uses standard Web technologies such as HTTP and URIs and extends them to share information in a way that can be read automatically by computers. This enables data from disparate sources to be connected and queried.

This structured repository will be a very useful contribution to the community of computer science and can be used by any application. In the future, IDEs would be equipped with such machine readable semantic datasets and would thereby be able to assist software engineers at a semantic level in managing the complexity of development, integration and maintenance projects

Using the structured knowledge base, our application will be able to make more intelligent choices about the nugget of knowledge that the information seeker is looking for and represent it in a manner that is easy to comprehend.

1

Introduction

"To start of something new, you need to be introduced to it first; and a true introduction never ends"

– Robert Anton Wilson

1.1 The Semantic Web

The Semantic Web, Web 3.0, the Linked Data Web, or the Web of Data represents the next major evolution in connecting information. It enables data to be linked from a source to any other source and to be understood by computers so that they can perform increasingly sophisticated tasks on our behalf.

Tim Berners-Lee first used the term in 1999 to describe the vision of machine-processable data on the Web, and today it is also the official name of the umbrella of W3C activity that is responsible for all of the relevant technology standards. [1]

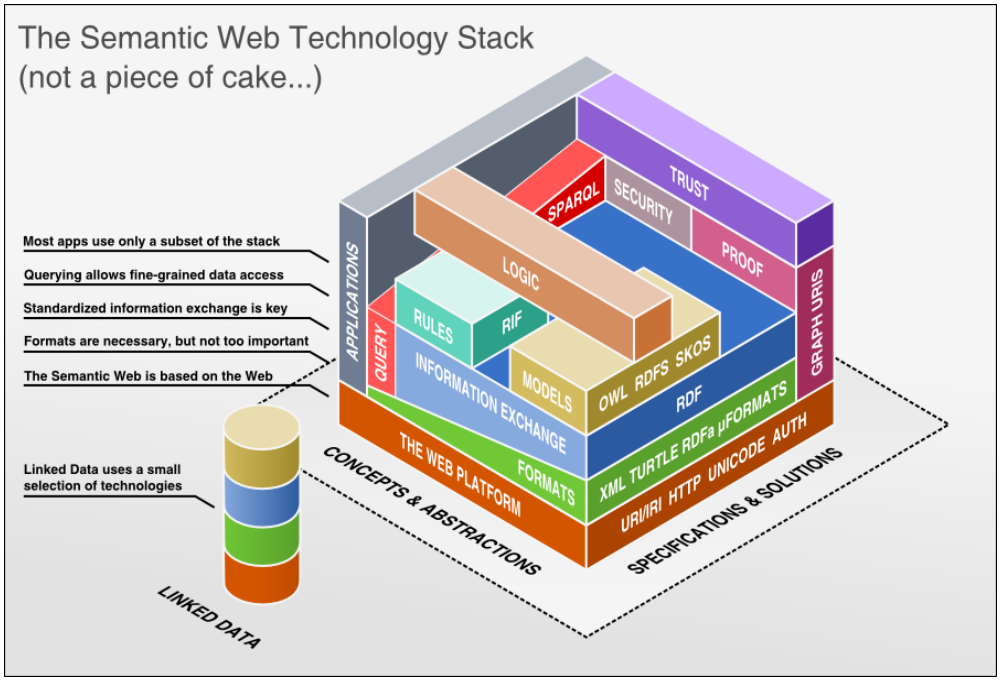


Fig 1.1 The Semantic Web - Not a Piece of Cake

Although *Semantic Web* is a term that has often been criticized as confusing, opaque, and academic, it does nonetheless capture two of the most important aspects of these technologies:

- **Semantic**: The meaning of the data is not only explicitly represented and richly expressive, but it also "travels" along with the data itself.
- **Web**: Individual pieces of data are linked together into a network of information, just as documents are linked together on the World Wide Web.

1.2 Linked Data and Linked Open Data

Seven years after introducing the term *Semantic Web*, Tim Berners-Lee gave us a new term: *Linked Data*. This was originally the name for a specific set of best practices for using Semantic Web technologies to publish data on the Web. These best practices emphasized creating links between different data sets and making the links between data items easy to follow without the need for any software other than a standard Web browser.

Because of the similarity between the terms *Linked Data* and *Linked Open Data*, you may find that people sometimes assume that using Linked Data technologies (or Semantic Web technologies, for that matter) requires that the information being represented be made publicly available. This is not the case, however, and it is important to make clear the distinction between explicitly representing information in a machine-processable manner (i.e., in the Semantic Web or via Linked Data) and publishing data sets on the Web for public consumption (i.e., Linked Open Data). Many enterprises are using Linked Data internally without ever exposing any of their data on the Web.

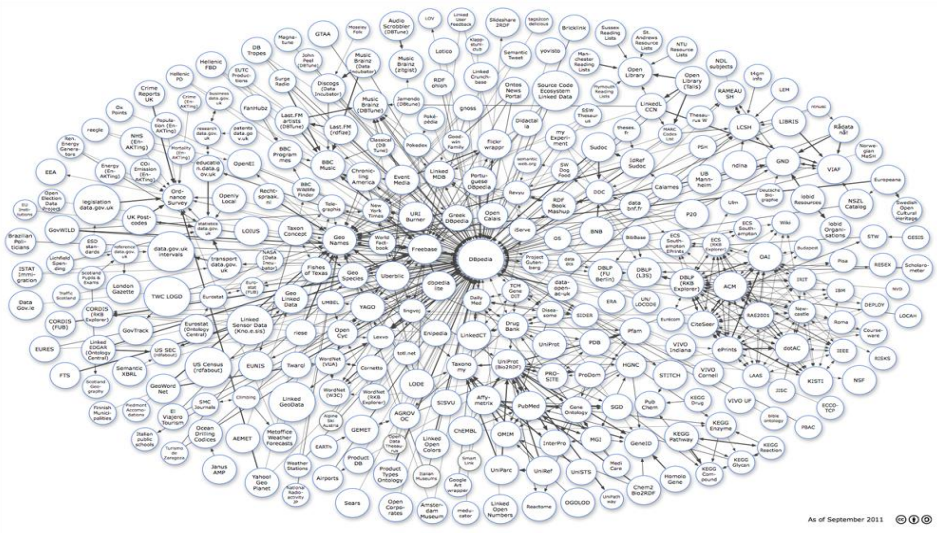
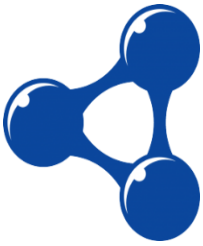


Fig 1.2.Linking open data cloud diagram

1.3 What Standards Apply to The Semantic Web?

From a technical point of view, the Semantic Web consists primarily of three technical standards:

- **RDF (Resource Description Framework)**: The data modeling language for the Semantic Web. All Semantic Web information is stored and represented in the RDF. RDF is the *data model* of the Semantic Web. That means that all data in Semantic Web technologies is *represented as* RDF. If you store Semantic Web data, it's in RDF. If you query Semantic Web data, it's RDF data. If you send Semantic Web data to your friend, it's RDF.



- **SPARQL (SPARQL Protocol and RDF Query Language)[18]**: The query language of the Semantic Web. It is specifically designed to query data across various systems.

- **OWL (Web Ontology Language)** The schema language, or knowledge representation (KR) language, of the Semantic Web. OWL enables you to define concepts composably so that these concepts can be reused as much and as often as possible. *Composability*means that each concept is carefully defined so that it can be selected and assembled in various combinations with other concepts as needed for many different applications and purposes.

Though there are other standards sometimes referenced by Semantic Web literature, these are the foundational three.

One way to differentiate a Semantic Web application vs. any other application is the usage of those three technologies.

1.4 Available Datasets

YAGO core	35GB
YAGO full	161GB
Person_data (from DBPedia) [19]	1.3GB
Datahub	213GB
DBPedia	51GB
DBPedia ontology	5GB
Freebase [20]	18GB
TIMBL (FOAF crawl-TIM Berners Lee)	39GB

Note that WordNet and a lot of DBPedia are included in YAGO.

Some basic statistics about this data (from http://gromgull.net/blog/2012/07/some-basic-btc2012-stats/):

- 1.85B triples all in all
- 1082 different namespaces are used
- 9.2M unique contexts, from 831 top-level document PLDs (Pay-Level-Domain, essentially data.gov.uk, instead of gov.uk, but livejournal.com, instead of bob.livejournal.com)
- 183M unique subjects are described
- 57k unique predicates
- 192M unique resources as objects
- 156M unique literals
- 152M triples are rdf:type statements, 296k types are used. Resource with multiple types are common, 45M resources have two types, 40M just one.

Though some of the above mentioned datasets have some triples from technical domain, an exclusive dataset comprising of purely technical knowledge does not exist. One such structured technical dataset will undoubtedly be a great contribution to the technical community.

2

Problem Definition

"A lack of clarity could put the brakes on any journey to success."
— Steve Maraboli, Life, the Truth, and Being Free

The aim of this project is:

1. To create a structured technical dataset (called "Technopedia")
2. To build an applicationthat uses the curated dataset thus highlighting the usefulness of Technopedia.

2.1 Objective

Software developers, test engineers, researchers and students face technical issues in the course of their work. The issues might be related to software compatibility, programming errors, syntax errors, or lack of familiarity with a new or evolving language. The solution to most of the above mentioned problems is to go through some technical documentation, Google it, look for a similar question on Stack overflow or to simply ask somebody who may or may not know the answer.

Unfortunately, information from these sources is very unstructured, distributed and difficult to comprehend. The process of going through the technical documentation is tedious and time consuming. What the person ideally wants is a quick "to-the-point" response to the question and also some related information which will assist him or her in grasping the solution.

The project(SWITH) addresses this problem - providing technical assistance and increasing the productivity of IT professionals by making use of structured and linked technical data. Linked data has the special quality of being highly interlinked and structured.

The solution to this problem involves semanticizing the relevant documentation and creating a linked open data store that will be effective to queried, thereby facilitating an easy visualization of relevant knowledge through an application. The understanding to a problem's solution is better when visualized. A visual representation is always more appealing and effective. This tool can also be used as a learning tool to sift through the technical information through the mind map provided.

SWITH is a proof of concept emphasizing that a structured i.e., semanticized technical dataset provides a better mine to search for information accurately. This is designed with the intent to enhance workplace productivity by trying to reduce the effort that goes into looking for the answer to a question in user guides, manuals and other documentation.

Apart from providing a proof of concept to the problem listed above, SWITH aims atusing it the next generation technical encyclopedia. The design is extendable and provides reusability of the data store by proving REST API access to it. When the dataset is expanded via crowd sourcing and enriched SWITH will be able to answer a wide range of technical questions accurately and save a lot of effort for the IT professionals.

3

Literature Survey

"Google' is not a synonym for 'research'."
— Dan Brown, The Lost Symbol

Query Interpretation- Background Study

The user of any search engine types a bunch of keywords in a search bar. The requirement is to answer the keyword query based on graph-structured data. This has emerged as an important research topic in the recent times. The prevalent approaches build on dedicated indexing techniques as well as search algorithms aiming at finding substructures that connect the data elements matching the keywords.

The paper [2]summarizes different factors affecting the effectiveness of the keyword search in relational databases. The authors have identified two other important factors: keyword proximity and keyword quadgrams that should be incorporated into the ranking function. Their experiments show that incorporation of each of these factors improves the effectiveness of the ranking function when included within the existing state-of-the-art information retrieval relevance ranking strategies for relational databases.

While trying to solve the core problem we were facing i.e. returning results from a database to users who do not know or even need to know the schema, we stumbled upon another earlier work [3]. In their paper, a keyword search method on relational databases finds joined tuples as answers, partitions them by interpretations of the query, and ranks those groups of answers. The ranking method focuses on finding answers derived from interpretations of the query that are similar to the interpretation in minds of ordinary users.

We explored the possibility of using a natural language query string instead of just keywords also. Our research led us to [4]. Using query language for dealing with databases is complex and requires schema knowledge as mentioned earlier. Usage of data existing in databases has a limitation of definite reports created in some pre implemented software. In this paper a

method is represented for building a "Natural Language Interfaces to Data Bases" (NLIDB) system. This system prepares an "expert system" implemented in Prolog which it can identify synonymous words in any language. It first parses the input sentences, and then the natural language expressions are transformed to SQL language.

We noted that by providing an expert system, they have encoded the hidden mystery of natural language. The fact that common words tend to have multiple meanings can lead to ambiguity. They have collected the required knowledge for this system from an individual who is experienced in natural language analysis, and embedded this knowledge into an expert system as a knowledge base. It finds the most similar entity name to the terms of input sentence based on searching this knowledge base. This paper is presenting the result of using an expert system beside common existing solutions for transforming natural language expressions to SQL query language.

The search for information on the Web of Data has become increasingly difficult due to its dramatic growth. Especially novice users need to acquire both knowledge about the underlying ontology structure and proficiency in formulating formal queries (e. g. SPARQL queries) to retrieve information from Linked Data sources.

When we searched for more specific studies on how to use SPARQL to query our Technopedia RDF dataset, we discovered an interesting paper [5]. The researchers have presented a natural-language question-answering system that gives access to the accumulated knowledge of one of the largest community projects on the Web – Wikipedia– via an automatically acquired structured knowledge base.

Key to building such a system is to establish mappings from natural language expressions to semantic representations. Their proposal was to acquire these mappings by data-driven methods – corpus harvesting and paraphrasing – and they also present a preliminary empirical study that demonstrates the viability of their method. An interesting example is given below.

The *Watson QA system* failed to correctly answer a substantial fraction of the questions in Jeopardy. One of these was:
"A big US city with two airports, one named after a World War II hero, one named after a World War II battle field."

Watson answered "Toronto"; the correct answer would have been Chicago. Interestingly, it would seem very easy to answer this question by executing a structured query over available knowledge bases or the existing and rapidly increasing set of Linked Data sources. All these sources are represented in the RDF format, and can be queried in the SPARQL language

The above question can be expressed as follows:

```
SELECT ?c WHERE {
        ?c hasTypeCity .
        ?a1 hasTypeAirport . ?a2 hasTypeAirport .
        ?a1 locatedIn ?c . ?a2 locatedIn ?c .
        ?a1 namedAfter ?p . ?p hasTypeWarHero .
        ?a2 namedAfter ?b . ?b hasTypeBattleField . }
```

where each search condition terminated by a dot is a triple pattern with three components – subject, predicate, object – some of which can be variables (starting with a question mark).

When the same variable appears in multiple triple patterns, the semantics is that of a relational join: results are bindings for the variable that simultaneously satisfy all predicates.

The arrival of huge structured knowledge repositories has opened up opportunities for answering complex questions, involving multiple relations between queried entities, by operating directly on the semantic representations rather than finding answers in natural-language texts. To reiterate the point, query languages for accessing these repositories are well established; however, they are too complicated for non-technical users, who would prefer to pose their questions in natural language. The YAGO-QA system gives access to the accumulated knowledge of Wikipedia via the automatically acquired structured knowledge base YAGO.

The main novelty of YAGO-QA lies in its mapping from natural language expressions in user questions to semantic representations (relations, classes, entities) in the knowledge base. Surface-text patterns are harnessed from a large-scale knowledge-harvesting machinery, and complemented this with a repository of question paraphrases gathered by a crowdsourcing approach.

The project necessitates the construction of SPARQL queries based on user-supplied keywords. So as to simplify and automate the querying and retrieval of information from RDF data sources, we consulted another paper [6].

Their approach utilizes a set of predefined basic graph pattern templates for generating adequate interpretations of user queries. This is achieved by obtaining ranked lists of candidate resource identifiers for the supplied keywords and then injecting these identifiers into suitable positions in the graph pattern templates. The main advantages of this approach is that it is completely agnostic of the underlying knowledge base and ontology schema, that it scales to large knowledge bases and is simple to use.

By tightly intertwining the keyword interpretation and query generation with the available background knowledge they are able to obtain results of relatively good quality. They also applied a number of techniques such as a thorough analysis of potential graph pattern so as to limit the search space and enable instant question answering.

A problem, however, beyond control is the data quality and coverage. Currently the evaluation of the technique described in that paper is still limited to 150M facts comprised by DBPedia, but due to the generic nature and efficiency of the approach they claim that it will extend it quickly to the whole Data Web. Another current limitation of their work is the restriction to two keywords. The rationale behind this was to restrict the search space of possible query interpretations, in order to return the most meaningful results to the user quickly in a compact form.

Lastly, in the paper[7], a novel keyword search paradigm for graph-structured data is described, focusing in particular on the RDF data model. We have listed several other papers to justify the reason for our selection of this particular paper. The approach followed in this work forms the core of our Query Interpreter and is therefore described in great detail in this chapter as well as further chapters.

Initially queries are computed from the keywords, allowing the user to choose the appropriate query, and finally, process the query using the underlying MYSQL[8] database engine. For the computation of queries in SQL, the top-k matching sub-graphs with the

lowest cost are utilized, including cyclic graphs. By performing exploration only on a summary data structure derived from the data graph, they achieve promising performance improvements compared to other approaches that do not make use of a summary graph.

4
Project Requirements

"Know your requirements and you are no more a child; you are set to go"

– Rebecca Pepper Sinkler

4.1 Product

The intended product from this project is a web based exploratory search assistant for technical information. The intelligent application will be able to interpret queries reasonably accurately. By using the structured knowledge base, will be able to make intelligent choices about the nugget of knowledge that the information seeker is looking for and represent it in a manner that is easy to comprehend. Further, it will also provide references to possible solutions elsewhere on the web.

4.2 Scope

Linked data set created will be limited to and rich with technical information. This data set is extendable and iterative. Hence it can be further enriched with the changing technology. The Technopedia is intended to be used by software developers and researchers. Many other helpful intelligent applications can be built upon the linked data set.

The assistant can also be used by the IT professionals, software developers, students or technology enthusiasts. The application would help solve software compatibility and error resolving queries.

4.3 Business Goals

Due to the ever expanding technology landscape, software developers are not always familiar with the languages and tools their job mandates them to deal with. They spend a significant amount of time reading through the unstructured documentation and finding the relevant information from it.

The assistant will help the software developers find answers to their queries more easily. In a broader sense, this will decrease product development time and boost productivity.

4.4 Product Features

Our contribution is an open linked dataset with around a million RDF triples. The data set is extendable and can be enriched incrementally. The lookup on the data set is efficient and consistent as it makes use of standard well known database SQL.

The search engine fetches information related to the technical queries. Easy to access and use web interface makes the assistant more user friendly.

4.5 User Classes and Characteristics

User roles are non-specific in our application. Broadly we can classify our users as software developers, IT professionals, students, teachers and enthusiasts. Our application will give an explorative and detailed insight into the domain of query. Although their special requirements may vary, our application will not distinguish between various roles norwill it offer specialized content to each of these classes.

4.6 Operating Environment

On the client side, the requirements are as follows:

- As the application developed is web–based, the system requirement on the user side is bare minimum. To specify explicitly, a user requires a modern browser and an internet connection.

On the server side, the requirements are as follows:

- A scalable server with load-balancing ability.
- A MySQL database server
- Query Engine and Web App.

While creating/ enriching Technopedia, the requirements are as follows:

- Linux platform with Linux Kernel 3.0 and above. We have used Ubuntu 12.04.
- Python 2.7 with multiple libraries described in detail in later sections.
- MYSQL database is used for indexing and storing the linked data set implicitly by RDFLib[9].

4.7 Assumptions and Dependencies

4.7.1 Assumptions

The assistant can answer queries related to technology and the data that it is aware of. The documentation of new technologies or the ones that have not been included in the data set could be added into the dataset as the data store is extendable.

The user must have a fair knowledge about computer science.

4.7.2 Dependencies

"Eggs are to Pythons as Jars are to Java..."

—http://peak.telecommunity.com/DevCenter/PythonEggs

- BeautifulSoup4[10][11]- The crawler makes use of the python library, Beautifulsoup- for pulling data out of HTML and XML files.

- Python 2.7 – with its extensive library and its development friendly syntax - seemed to be the obvious choice of language for implementation.

- Nltk [12]- a suite of libraries and programs for symbolic and statistical natural language processing (NLP) for the Python programming language.

- Simplejson- a simple, fast, complete, correct and extensible JSON [13] encoder and decoder for Python 2.5+ and Python 3.3+.

- Networkx [14] - a python package for creating and manipulating graphs and networks.

- Urlparse-This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

- Matplotlib [15]- strives to produce publication quality 2D graphics for interactive graphing, scientific publishing, user interface development and web application servers targeting multiple user interfaces and hardcopy output formats.

- Flask- a microframework for Python based on Werkzeug, Jinja 2.

- Urllib- a high-level interface for fetching data across the World Wide Web.

- Feedparser- Parse Atom and RSS feeds in Python.
- Rfc3987[16]- provides regular expressions according to RFC 3986 "Uniform Resource Identifier (URI): Generic Syntax" and RFC 3987 "Internationalized Resource Identifiers (IRIs)", and utilities for composition and relative resolution of references.

- Difflib [27]- Helpers for computing deltas. This module provides classes and functions for comparing sequences.

- Rdflib[9]- pure Python package providing the core RDF constructs. The rdflib package is intended to provide core RDF types and interfaces for working with RDF.

- PyMySQL - library to interface MySQL database to our application.

- Bing API – search engine results

- Epydoc [17]- a tool for generating API documentation for Python modules, based on their docstrings.

5

System Requirements

"Give me six hours to chop down a tree and I will spend the first four sharpening the axe."

— Abraham Lincoln

5.1 Interface Requirements

5.1.1 Technopedia Interface

The Technopedia should be accessible to the users for retrieving data. Users should be able to access one of the subjects, objects and predicates. Additionally, the users should also have access to complete triples and literals.

5.1.2 Application Interface

The application should have an interface to accept a keyword query from the user as a string.

The implementation of the interface should return the relevant facts.

5.2 Functional Requirements

5.2.1 Typical Use-Case Scenarios:

Purpose: Mr Tony Stark wants to obtain information about a particular technology subject.

Preconditions: Mr Tony Stark is connected to the internet and has access to the web application.

Postconditions: Mr Tony Stark obtains the required information in the desired format.

Scenario-1

- Mr. Tony Stark issues a get request with a required response type and a particular subject, predicate or object.

- The application directly queries for relevant information from the dataset.
- The results are returned as a response in JSON format.

Scenario-2

- Mr Tony Stark opens the web app and enters a query string.
- The application looks for relevant information from the dataset and also the other APIs that have been plugged in for enhanced results.
- The results are displayed in the window in a user friendly and appealing manner.
- Tony reads through to see if what he is looking for is there.
- He studies the mind map of the ontology.
- He browses through the other search results returned.
- He looks at the related questions that have been discussed on a Q & A forum that our application also provides.
- His doubt is resolved. He is completely satisfied and continues his work!

5.3Non-Functional Requirements

5.3.1 Performance Requirements

The website must have a quick response time of less than 30 seconds and therefore, the underlying dataset should be relatively selective and clean.

5.3.2 Software Quality Attributes

- Care has to be taken to ensure the correctness and cleanliness of the data.
- Information retrieved by the engine should be relevant to the search query.
- The data store is extendable and scalable.

5.4 Design and Implementation Constraints

Data

- The Technopedia dataset should be of a linked and open nature.
- It should be extendible.

- The portion of the dataset obtained by filtering DBPedia, Freebase, etc. should not have RDF quads that do not belong in the set.
- Complete frames of subjects and objects are required to be present.
- The portion of the dataset fetched by crawling, parsing and extracting RDF quads from HTML pages containing technical information should be accurate.
- The URIs referred to in the quads must point to valid unique locations that contain actual descriptions as far as possible, according to the standard.

Application

- The web application will receive a string query from the user as keywords.
- The website must have a quick response time of less than 30 seconds and therefore, the underlying dataset should be relatively selective and clean.
- Provide API access to the Technopedia dataset so that other applications can utilize it in innovative and exciting ways.

6
System Design

"When you want to know how things really work, study them when they're coming apart."
— William Gibson, Zero History

The project is divided into four components as shown below.

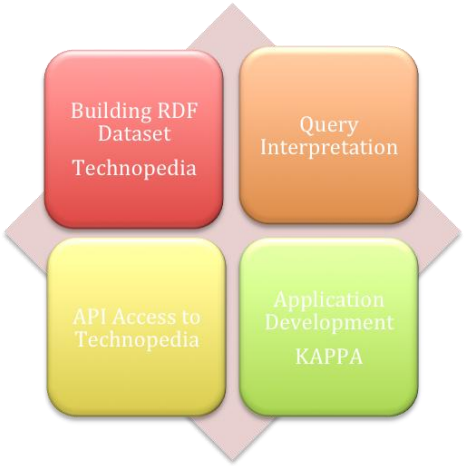


Fig. 6.1 Project Components

6.1Building RDF Dataset - Technopedia

A technical RDF dataset is built by extracting the technical data from existing RDF datasets (such as Freebase and DBPedia) and also by semanticizing technical documentations (such as JavaDocs[21]).

6.2 Technopedia API Creation

A REST (Representative State Transfer) API (Application Programming Interface) to access data from the Technopedia knowledge base is provided so that other developers can directly utilize our linked open data in their creative applications. They can aggregate their data from a variety of linked open data repositories to draw their own inferences from the data.

6.3 Query Interpretation

Based on the keywords entered as a user query string, the engine can return a set of system queries. These are like suggestions to show how the entered keywords were interpreted. An

algorithm can then appropriately choose the query that best describes what he/she was searching for.

6.4 Application Development (Kappa)

The application that is deployed must have a user friendly interface. Results from a variety of sources like search engines and from QA forums are also rendered on the screen. A mind map of the concepts is also drawn on a canvasto facilitate visualization of knowledge.

The application Kappa which makes use of the semanticized technical documentation (“named Technopedia”)is an explorative fact search engine that provides,

1) Relevant facts from Technopedia.
2) A visual mind map representation of the facts obtained.
3) Relevant search results from Bing.
4) Similar questions posted on Stackoverflow.

6.5Architecture

The architecture encompassing the four components is pictorially represented in Fig. 6.2.

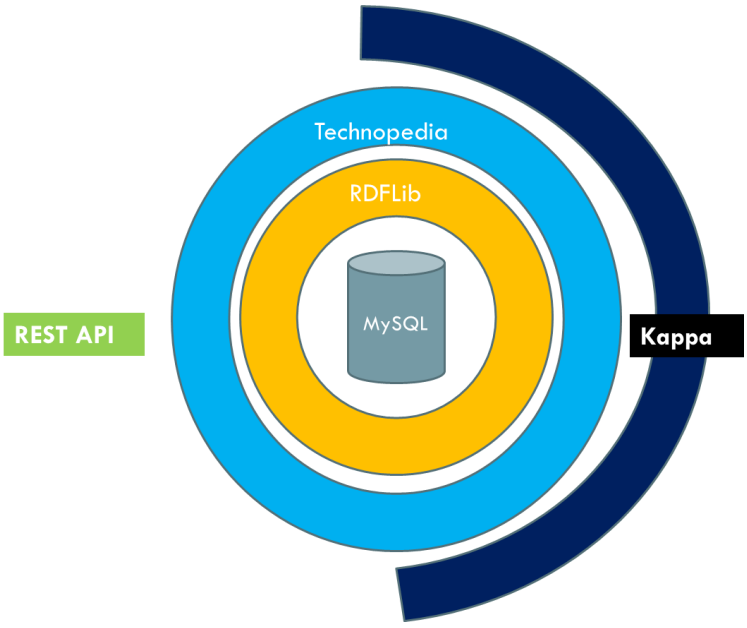


Fig. 6.2 System Architecture

The core part of the architecture has the MySQL data store. The collected dataset is loaded into a MySQL database by using the RDFLib Python library.

This is accessed via RDFLib - a python library that provides many functions to operate on the dataset.

Technopedia is a wrapper around the dataset which makes use of RDFLib functionalities to provide access to data and also provides some helper functions that can be useful to the application. It effectively conceals the RDFLib functions and provides user-friendly functions to access (and not manipulate) the data.

REST APIs are provided to make Technopedia accessible to any application over the web.
It also acts as a Python library which can be used by any other python application directly.

Kappa is an application which uses the Technopedia library to access the dataset and provides a keyword based search.

7
Dataset: Technopedia

"We're entering a new world in which data may be more important than software."
– Tim O'Reilly

7.1 Extraction of technical data from existing RDF datasets

Technical data already present Freebase and DBPedia were identified by systematically making note of different types/domains of technical data available.

The relevant information in the form of quads was then extracted from the complete Freebase and DBPedia dumps by recursively 'grep'ing for those types. The subset of the complete dump so obtained was then cleaned by using other UNIX shell commands: sort, uniq and cut.

A peek into the obtained dataset is given below.

Type	No. of Properties	No. of Instances	Description
Computer	10	671	Any computer, including desktop machines, games consoles and portable devices.
Computer Designer	1	45	A person, company or the like which did the high-level design for a computer.
Computer Emulator	3	230	A computer emulator is a system, usually implemented in software that is designed to simulate the behavior of a computer.
Computer Manufacturer/Brand	2	101	A company, organisation or the like responsible for the manufacturing and/or sale of a computer to the public.

Computer Peripheral	5	85	A computer peripheral: a device which is attached to a computer to provide additional functionality, but which (in general) requires the computer to work.
Computer Peripheral Class	6	52	A generic class of peripherals, for example "joystick" or "light gun". Specific peripherals should use the Computer Peripheral type.
Computer Processor	5	293	The device which acts as the central core in a computer.
Computer Product Line	0	9	The line or brand name of related computer-based products from a manufacturer such as the Apple II (Apple II, Apple IIe, etc.) or a smartphone like the BlackBerry.
Computer Scientist	0	876	A person involved in research relating to computers, including both hardware and software.
Computing Platform	1	205	Computing platform is a category of devices that run computer programs.
Content License	0	29	Type of license that denotes the rights and restrictions of usage of the content.
File Format	13	3,559	A way that data is encoded in a computer file.
File Format Genre	1	60	Categorization of the file format.
HTML Layout Engine	1	7	A software component which can be used to cause HTML to be displayed on a screen or other output device.

Internet Protocol	1	343	The Internet Protocol (IP) is the principal communications protocol used for relaying datagrams (packets) across an internetwork using the Internet Protocol Suite. This type captures the various forms of IP used by devices and programs communicating over some sort of network.
Operating System	6	568	The software on a computer which is responsible for direct communication with the hardware and other low-level issues.
Operating System Developer	1	110	The person, company, organisation or the like responsible for the development of an operating system.
Processor Manufacturer	1	52	The company or other organisation responsible for the manufacture of a computer processor.
Programming Language	8	1,181	A language used to give instructions to a computer.
Programming Language Designer	1	170	The person, company, organisation or the like responsible for the high-level design of a programming language.
Programming Language Developer	1	52	The person, company, organisation or the like responsible for the actual implementation of a programming language, as opposed to the high-level design (although in many cases these will be the same).
Programming Language Paradigm	1	55	A fundamental style of computer programming.

Protocol Provider	0	45	The organization responsible or created the protocol (rules determining the format and transmission of data for computers).
Software	10	8,862	Any software which runs on a computer, excluding games (which should use the Video Game type instead).
Software Developer	1	1,574	Any entity (person, company or anything else) which has developed computer software (other than games; they use the Computer Game Developer type).
Software Genre	3	520	The general top-level description of the function of a piece of software.
Software License	2	133	The legal terms under which a piece of software is released.
Web browser	2	64	A piece of software which can be used to browse the World Wide Web.
Web browser extension	1	18	A piece of software which uses a browser's plug-in architecture in order to extend the capabilities of the browser in some way.
OS Compatibility	4	162	A compound value type detailing which versions of an operating system run on a specific computer.
Software Compatibility	4	8,425	A compound value type detailing which operating system versions a specific piece of software can run on.

Table 7.1 Freebase- Computers

7.2 Semanticizing Javadocs

The Java 7 SE API Documentation was considered for semanticizing for two reasons:

1. Due to widespread use of Java in the industry and in the academic field, availability of a structured documentation would be of great help.
2. The semi-structured nature of Java API documentation allows extraction of clean and accurate facts relatively quickly.

The API documentation Java 7 SE was obtained from the official website and parsed using a Python Library called BeautifulSoup4. The parser delivered the information in layers:

1. Package Information
2. Class Information
3. Method Information.

The information so obtained was analysed and list of predicates which relates the entities were formulated. By using these predicates, the information from Javadocs was transformed into an RDF quad structure.

A peek into the obtained information is given below.

Type	# of Properties	# of Instances	Description
Packages	1	591	All the packages present in Java 7 Official API Documentation
Classes, interfaces, annotations, errors, exceptions, enumerations	6	15488	All the classes, interfaces, annotations, errors, exceptions and enumerations present in Java 7 Official API Documentation
Methods	1	252196	All the methods present in Java 7 Official API Documentation

Table 7.2 Javadocs Statistics

Fig. 7.1 Screenshot of a small part of Technopedia shows the quad structure

7.3 Loading to Datastore

The Technopedia obtained had 1.2 million plus quads. Given below are some statistics:

	Original Size	Required Size	Required #Quads
Freebase	49GB	123.6MB	740149
DBPedia	44GB	185.2MB	433834
Javadocs	<1GB	60.8MB	268275

Given the huge size of the dataset, it was necessary to load the data into a fast and efficient datastore for quick access by the application. Accessing the data directly from the file would not be possible for quick responses.

A number of datastores including Jena [22], Sesame[23], Virtuoso[24], 4store[25] and relational databases such as MySQL[8], SQLite[26] were compared for the purpose of loading the dataset. After a detailed study, the choice of stores boiled down to Jena and MySQL.

It was observed that though the loading in Jena was faster than MySQL, the access was much slower. Hence, MySQL was chosen to be the datastore.

Once loaded, the data would be accessed from an application; hence, it was necessary to create a proper database schema prior to loading.

Therefore, a Python library – RDFlib, was selected to handle the datastore and data access.

RDFlib is a Python Library to handle RDF data. The library contains an RDF/XML parser/serializer that conforms to the RDF/XML Syntax Specification (Revised). The library also contains both in-memory and persistent Graph backend such as MySQL, SQLite, etc. RDFlib automatically creates a database schema and loads the data which facilitates easy access to the data.

7.4 Technopedia API

The accessing of Technopedia through RDFlib, though easy is not straight forward; it requires using the specific term objects provided by RDFlib. This arrangement would not be helpful to an application willing to access the Technopedia. Further, RDFlib allows addition/deletion/modification of data being accessed which should not be exposed to the application. Hence, a wrapper module which wraps around the RDFlib constructs and abstracts the functionalities was designed and implemented.

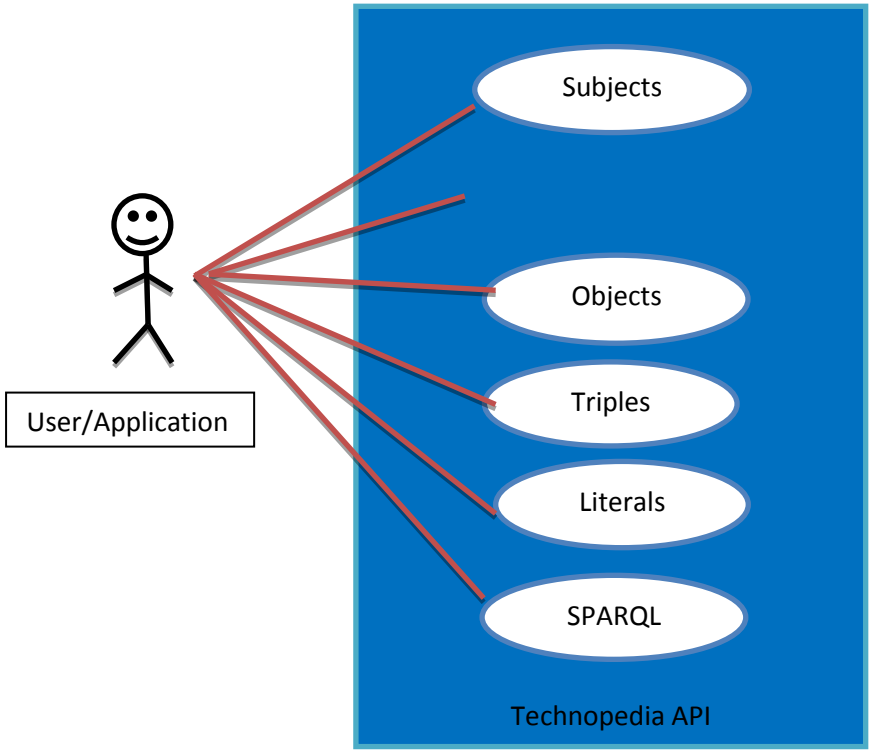


Fig. 7.2 Use case diagram for Technopedia API

The Technopedia API module allows the input to be in form of a string as it internally converts it to the form required by RDFlib, obtains the result and reconverts it to string format to be returned to the user/application.

See Appendix A for complete Technopedia API documentation.

7.4.1 REST API Access

The Technopedia API would be hosted on a web platform for community use. Hence, a REST API access was implemented to enable accessing the data over Web.

The user/application can simply issue an HTTP GET request with the function name as a value of variable “responsetype”. The arguments of the function also become the variables in the request.

Eg:subjects(object=”Driver”) is technopedia.com/?responsetype=subjects&object=Driver

Fig. 7.3 Accessing Technopedia through REST API

8
Application: Kappa

"Knowledge without application is like a book that is never read"
– Christopher Crawford, Hemel Hempstead

SWITH's ultimate goal is to be a handy helper application that provides technical assistance to the developers. The aim is to enhance workplace productivity by reducing time spent sifting through lengthy documentation.

Ideally, given keyword query, the application should return a set of relevant facts. Our application – *kappa* – attempts to do the same by performing a **graph based explorative keyword search** over a well-structured RDF dataset.

8.1 RDF Data Graph Design

The RDF data is considered as a graph where subjects and objects are nodes and the predicates are directed edges between the nodes.

The data graph has the following types of nodes:

1. C-vertices: Nodes representing the type of an entity (classes).
2. E-vertices: Nodes representing entities.
3. V-vertices: Nodes representing Literal values.

Additionally, "BNode" and "Thing" are considered as special types of C-nodes representing Blank nodes and Entities which do not belong to any particular class respectively.

The graph has the following types of edges (predicates):

1. A-edges: Edges between E-vertices and V-vertices.
2. R-edges: Edges between E-vertices.
3. "Type": Edges between E-vertices and C-vertices.

8.2 Data structures

8.2.1 Graph Element

The graph element (GE) data structure obtains data from Technopedia and maintains information in the form of graph elements.

It encapsulates the meta-data about data elements such as the type, sub-type, parents, cursors, etc.

It provides an interface to deal with the data as required by any Graph Exploratory algorithm; provides methods to access neighbours, weights, type of graph elements, etc.

See Appendix B for complete documentation of this data structure.

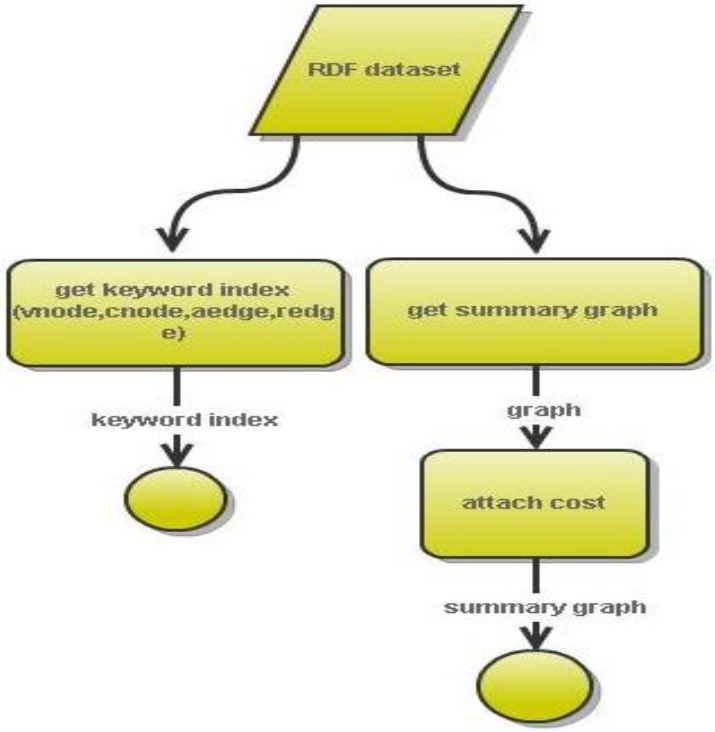


Fig. 8.1 Data Structures

8.2.2 Keyword Index

Keywords entered by the user might refer to data elements (constants) or structure elements of a query (predicates). From the data graph point of view, keywords might refer to C-vertices (classes) or V-vertices (data values) and edges. We assume that keywords will not map E-vertices as they are either URIs or BNodes and the user is unlikely to input the URI as a query term.Therefore a mapping between the set of keywords and the graph element referred to by them is maintained

TheC-vertices and the edges are in the form of a URI and hence they have to be parsed to obtain keywords from synonymous with them. We have implemented the function extract_keywords_from_uri() and the supporting helper functions for this purpose.

8.2.3 Summary Graph

The graph index is used for the exploration of substructures that connect keyword elements. Exploration might be very expensive when the data graph is large. To solve this problem, a summary graph which intuitively captures only the relation between classes of the entities is constructed. This information is sufficient to obtain queries which can then be run on the complete dataset.

A summary graph consists of C-nodes and R-edges. Each R-edge connects those C-nodes to which the E-nodes connected by the edge in the original graph belong.C-vertices are preserved in the summary graph because they might correspond to keywords and are thus relevant for query computation.A-edges and V-vertices have not been considered in the construction of the summary graph. By definition, a V-vertex has only one edge, namely an incoming A-edge that connects it with an E-vertex. This means that in the exploration for substructures, any traversal along an A-edge ends at a V-vertex. Thus, both A-edges and V-vertices do not help to connect keyword elements. They are relevant for query computation only when they are keyword elements themselves. In order to keep the search space minimal, the summary graph is augmented only with the A-edges and V-vertices that are obtained from the keyword-to-element mapping during query-interpretation.

Graph Element Cost

A cost is associated with each graph element of the summary graph based on the popularity of the element in the original data graph. The following cost function captures the"popularity" of an element of the summary graph, measured by the relative number of data elements that it actually represents.

$$c(v) = 1 - \frac{|vagg|}{|V|}$$

where, $|v_{agg}|$ = No. of E-vertices that have been clustered to a C-vertex

|V| = Total No. of vertices in the summary graph.

$$c(e) = 1 - \frac{|eagg|}{|E|}$$

where, |eagg| = No. of original R-edges that have been clustered to a corresponding R-edges ofthe summary graph.

|E| = Total No. of edges in the summary graph.

8.3 Query Interpretation

The query interpretation component of the application understands the keywords entered by the user and returns the best k queries that the user might have meant to ask. The approach used for keyword search is based on graph-structured data, RDF in particular.

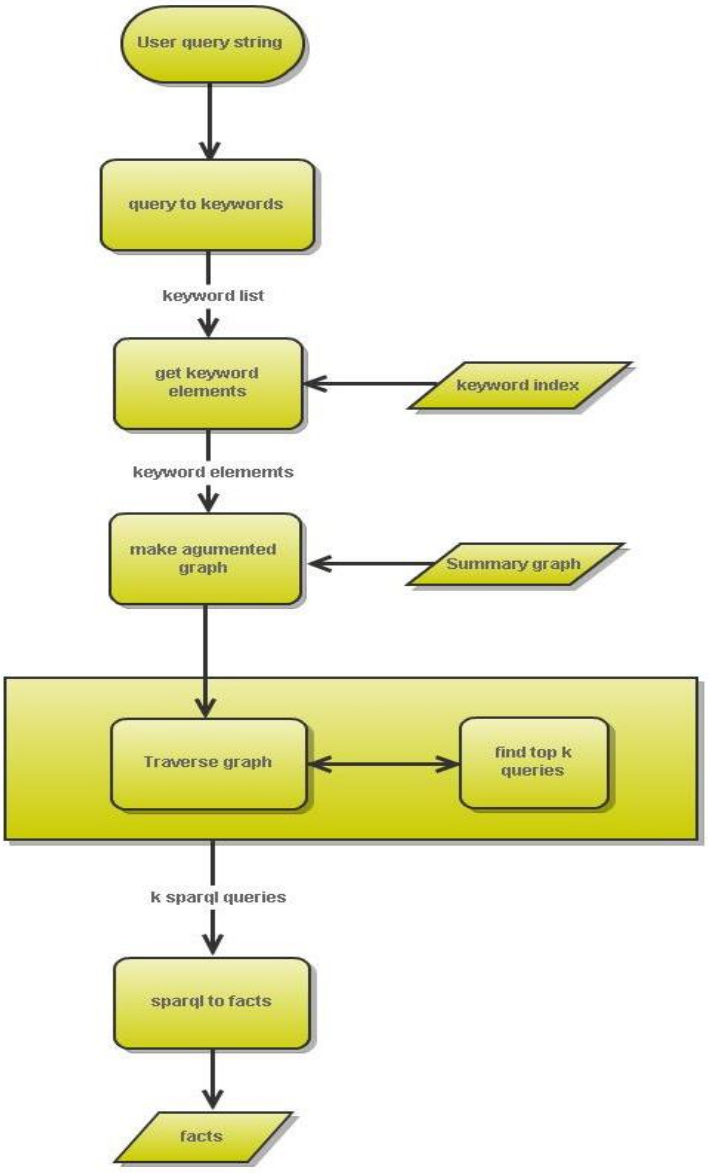


Fig. 8.2 Query Interpretation and Fact Generation

The basic idea is to map keywords to data elements (keyword elements) or in simpler words a graph element.For keyword mapping, the keyword index is used to obtain a possibly overlapping set of graph elements for every given keyword. Every element in this set is representative for one or several keywords. These elements are added to the summary graph to make an augmented summary graph.

Using this, a sub graph is constructed by searching for substructures on the data graph that connect the keyword elements. The top-k substructures are computed on the basis of a scoring function. These substructures are in turn converted into queries. The queries are run on the entire dataset and the results obtained are polished to extract readable facts.

8.3.1 User query to keyword mapping

Query to keyword mapping module takes in the keyword query, processes it and finds matches to the keywords in the keyword index built using the data. Firstly, the query string is pre-processed to remove the stop words. This makes use of the nltkcorpus [12] which contains stop words. The remaining essential words each are matched with all the entries in the keyword index using the python package called difflib [27]. This module provides classes and functions for comparing sequences. The sequence matching function in difflib performs block matching and returns triple describing matching.

Scoring functions

There are two scoring functions devised for rating the match.

The first one makes use of the inbuilt function in difflib that performs scoring based on the formula below,

$$\textbf{Score} = 2.0 * \frac{M}{T}$$

where,

T is the total number of elements in both sequences

M is the number of matches

Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

The second one considers the length of match of each block, disregards all the matching length below three to exclude the prefix, suffix and single letter match. The length of matches of these selected blocks is cumulated. A ration of this sum is taken with the length of the query word being matched. The formula below illustrates the same,

$$\textbf{Score} = \sum_{k=0}^{n} B / L$$

where,

n is number matching blocks

B is the length of matching blocks /B> 3

L is length of the query word

A Threshold value of 0.5 and 0.8 are set for the first and second function respectively.
The highest matching score key word is mapped with the query word.

8.3.2Keywords to KeywordElement Mapping.

The keywords are mapped to data elements (constants) or structure elements of a query (predicates).A keyword-element map is in fact an IR engine, which lexically analyses a given keyword, performs an imprecise matching, and finally returns a list of graph elements.After obtaining the keywords, a list graph elements corresponding to each keyword is obtained. (get_keyword_elements())

8.3.3Augmented Graph

An augmented graph is constructed by augmenting the V-vertices and A-edges from the list of keyword elements to summary graph. The augmented graph now has all the graph elements corresponding to the Keyword list. (make_augmented_graph())

8.3.4Search for Minimal Subgraph

The graph is explored forward using the mapped keyword-elements of the graph as the starting points. As the graph is explored, in each step all the combinations of the explored paths are reviewed to see if they could form a substructure to connect all the keyword elements. This process is repeated to find the best k substructures.

Input and DataStructures

The input to the algorithm comprises the elements of the graph summary G andthe keyword elements K = (K_1... K_m) where each $K_i$corresponds to the set of data elements associated to $keyword_i$(which have been retrieved using the keyword index). Further, k is used to denote the number of queries to be computed. The maximum distance $d_{max}$is provided to constrain the exploration to neighbours that are within given vicinity. The cursor concept is employed for the exploration. In order to keep track of the visited paths, every cursor is represented as C(n, k, p, d,w), where n is the graph element just visited, k is a keyword element representing the origin of the path captured by c and p is the parent cursor of c. Using this data structure, the path between n and k_i captured by Ccan be computed through the recursive traversal of the parent cursors. Besides, the cost w and the distance d (the length) is stored for the path. In order to keep track of information related to a graph element n and the different paths discovered for n during the exploration, a data structure of the form (w, (C_1, ...,C_m)) is employed, where w is the cost of n and Ci is a sorted list of cursors representing paths from $k_i$to n. For supporting top-k (see next section), LGis used as a global variable to keep track of the candidate subgraphs computed during the exploration

Initialization

Similar to backward search, the exploration starts with a set of keyword elements. For each of these, cursors with an empty path history are created for the keyword elements and placed into the respective queue $Q_i \in$ LQ. During exploration, the "cheapest"cursor created so far is selected for further expansion. Every cursor expansion constitutes an exploration step, where new cursors are created for the neighbours of the element just visited by the current cursor. At every step of the exploration, top-k is invoked to check whether the element just visited is a connecting element, and whether it is safe to terminate the process.

Exploration

At each iteration, a cursor c with the lowest cost is taken from $Q_i \in$ LQ.Since $Q_i$is sorted according to the cursors' cost, only the top element of each Q has to be considered to determine c. In casethat the current distance c.ddoes not exceed the parameter d_{max}, c is first added to the corresponding list $C_i$of n, the graph element associated with c. This is to mark that n is connected with c.kthrough the path represented by c. During top-k processing, these paths are used to verify whether a given element n is a connecting element.

```
Algorithm 1: Search for Minimal Matching Subgraph
Input: k; dmax ; GK = V E; K = (K1 , ..., Km ).
Data:c(n, k, p, d, w); LQ = (Q1 , ..., Qm);n(w, (C1, ..., Cm )); LG (as global var.).
Result: the top-k queries R

// add cursor for each keyword element to Qi ∈ LQ
foreach Ki ∈ K do
  foreach k ∈ Ki do
        Qi.add(newCursor (k, k, ∅, 0, k.w));
  end
end

while not all queues Qi ∈ LQ are empty do
  c ← minCostCursor(LQ);
  n ← c.n;
  if c.d<dmax then
        n.addCursor(c);
        // all neighbors except parent element of c
        Neighbors ← neighbors(n) \(c.p).n;
        // no more neighbours!
        if Neighbors != ∅ then
          foreach n ∈Neighbors do
          // cyclic path when n already visited by c
                if n does not belong to parents(c) then
                  // add new cursor to respective queue
                  Qi.add(new cursor (n, c.k, c.n, c.d +1, c.w + n.w));
                end
          end
        end
        Qi.pop(c);
        R ← Top-k(n, LG , LQ, k, R);
  end
end
return R;
```

Then, the algorithm continues to explore the neighbourhood of n, expanding the current cursor by creating new cursors for all neighbour elements of n (except the parent node (c.p).n that we have just visited) and add them to the respective queues Qi.Also, cj.nshould not be an element of the current path, i.e. it is not one of the parent elements already visited by ci. Such a cursor would result in a cyclic expansion. The distance and the cost of these new cursors are computed on the basis of the current cursor c and the cost function as discussed earlier. Since this cost function allows the contribution of every path to be computed independently, the cost of a new cursor is simply c.w + n.w, where n is the neighbour element for which the new cursor has been created.

8.3.5Top-k Query Computation

The graph is explored forward using the mapped keyword-elements of the graph as the starting points. As the graph is explored, in each step all the combinations of the explored paths are reviewed to see if they could form a substructure to connect all the keyword elements. This process is repeated to find the best k substructures.

Algorithm 2: Top-k Query Computation
Input: n, LG, LQ, k, R.
Output: R.

```
if n is a connecting element then
   // process new subgraphs in n
   C ← cursorCombinations(n);
   foreach c ∈ C do
         LG.add(mergeCursorPaths(c));
   end
end

LG ← k-best(LG);
highestCost ← k-ranked(LG);
lowestCost ← minCostCursor(LQ).w;
if highestCost<lowestCost then
 foreach G ∈ LG do
         // add query computed from subgraph
         R.add(mapToQuery(G));
   end
   // terminates after top-k have been computed
   return R;
end
```

Scoring

To find the best k substructures the graph elements need to be scored. There is a score associated with the nodes and edges of the graph. The cost of the substructure is the cumulative sum of the weights associated with all the graph elements belonging to the sub graph. There are various cost functions that can be used to assign costs to the graph elements, distance based or popularity based. The best suited function will be used.

Top-k processing will lead to the early termination after obtaining the top-k results, instead of searching the data graph for all results. It uses the basic idea that originates from the Threshold Algorithm (TA). Top-k results have been obtained if the highest cost of the candidate for the top-k is found to be lower than the lowest cost of the remaining candidates.

8.3.6 Query Mapping

The subgraphs as computed previously aremapped to conjunctive queries. Note that exploration has been performed only on the augmented summary graph. Thus, edges of subgraphs are either A-edges or R-edges.

A complete mapping of such a subgraph to a conjunctive query can be obtained as follows:

- **Processing of Vertices**:

  Labels of vertices might be used as constants. Thus, vertices are associated with their labels such that constant(v) returns the label of the vertex v. Also, vertices might stand for variables. Everyvertex is therefore also associated with a distinct variablesuch that var(v) returns the variable representing v.

- **Mapping of A-edges Edges:**

  A-edgesare mapped to two query predicates of the form:

  var(v1)<type predicate> constant(v1)

  var(v1)<edge>constant(v2).

  Note that v1 denotes a class and constant(v1) returns a class name.

  In case v2 = value, edgeis mapped to the predicates:

  var(v1)<type predicate> constant(v1)

  var(v1)<edge>var(value).

  If v1 is a Blank Node or has no class associated with it, the predicate form with <type> information is not output.

- **Mapping of R-edges:**

  R-edges are mapped to three query predicates of the form:

  var(v1)<type predicate> constant(v1)

  var(v2)<type predicate> constant(v2)

  var(v1)<edge>var(v2)

  If v1/v2 is a Blank Node or has no class associated with it, the predicate form with <type> information is not output.

8.3.7 Generate Facts

The results returned by the SPARQL engine are in JSON format. They provide a binding from variables to URI. The SPARQL query is used as a skeletal structure to build facts. The useful query in the form of subject predicate object <spo> is used. The <spo> like label or membership to a URI are disregarded.

Subject, predicate and object can be one of the below three:

1. Variable
2. URI
3. Literal

They are handled in the following ways,

- Replacing the variables with their respective binding provided in the JSON result. The binding URI is replaced with its respective label.
- The predicates are converted to keywords. These keywords represent what the predicates means to say in English.
- The literals are left as such.

The resulting fact is a semi English statement that can be easily understood by IT professionals.

8.4 Web Application

Given the user query, the client issues a get request to kappa and obtains facts and the corresponding URIs which is displayed.

> *"Due to the size of graphs and the complexity of questions, visualization is the natural tool to understand what is going on."*

The relation between the various elements is also displayed as a mindmap (graph based visualization) for easy understanding.

Apart from the facts from Technopedia, results relevant to the user query are also fetched from Bing and Stackoverflow and displayed which provides the user with extra information at the same place and also helps highlighting the usefulness of Technopedia.

9
Integration

"Design is not making beauty; beauty emerges from selection, affinities and integration."

– Louis Kahn

SWITH hastwo major modules namely:

1. Technopedia (Dataset access)
2. Kappa(Query interpretation+Application)

The two modules are largely independent of each other and communicate only to access the Technopedia RDF dataset.

The interactions between the modules and external components are:

1. Technopedia – Kappa: Indexing
2. Kappa – Technopedia: RetrieveFacts
3. Kappa – Bing: Retrieve relevant Bing and Stackoverflow results.

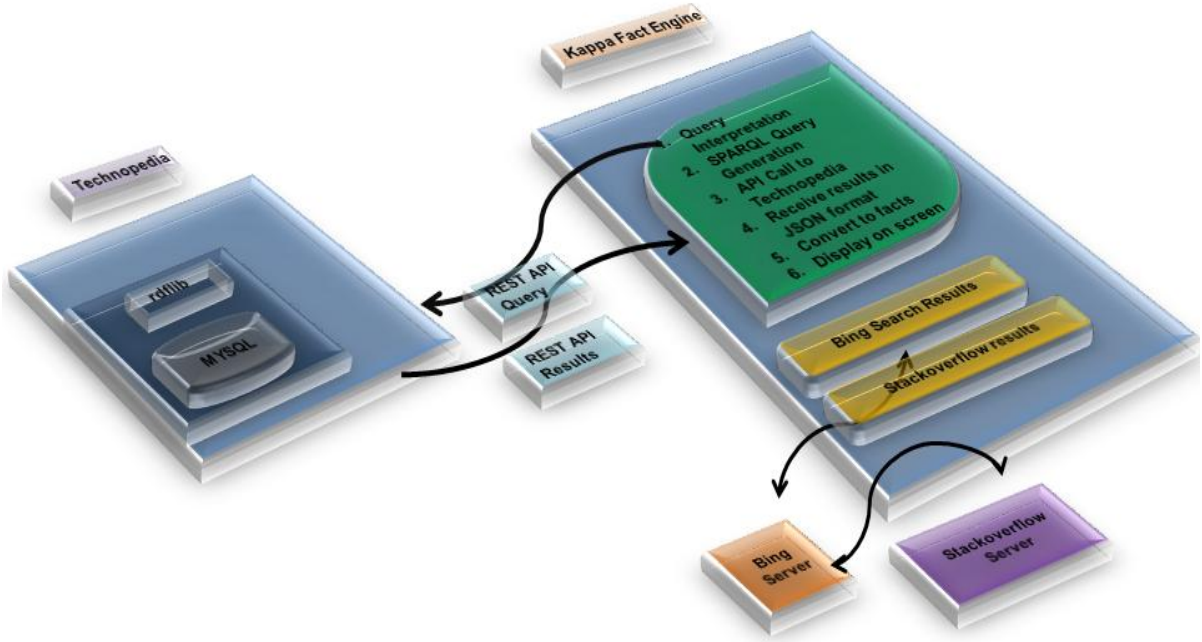


Fig. 9.1 Module Integration

9.1 Technopedia – Kappa: Indexing

A keyword index and summary graph is constructed from the data set. A SQL data store is used for storing the data. The data needed for the construction of keyword index and summary is obtained from the data store using a REST API. The REST API obtains results from RDFlib, polishes the results and returns the required part of the result in an appropriate format.

9.2 Kappa – Technopedia:Retrieve Facts

Kappa provides a web interface for the users to enter a keyword query. This keyword query is pre-processed and processed to obtain a mapping to the keyword index built. An augmented summary graph is constructed using the mapping obtained. Query interpretation provides the SPARQL queries. These queries are then executed by accessing the dataset through the SPARQL endpoint provided by the Technopedia API.

9.3 Kappa – Technopedia: Retrieve relevant Bing and Stackoverflow results.

The keyword query entered by the user is passed to the Bing search engine to obtain top-5 results from Bing and Stackoverflow.

10

Testing

"Program testing can be used to show the presence of bugs, but never to show their absence!"

– Edward Dijkstra

The system was tested extensively at every step of the development.

While the testing of the modules, independently, during the development phase guided the further development and fine – tuning of the respective modules, the testing the whole system after integration was more crucial and important.

10.1 Modules Tested

- Technopedia
    - Data.py
    - Kappa.py
- Restaccess.py
- Webapp.py

10.2 Test Strategy

Since a qualitative study of the effectiveness of Kappa needed to be done, we approached people to try it out and give us feedback in addition to our own testing.

As a complementary measure for testing, we also carried out testing of individual units, components and the interactions among them from a broad perspective.

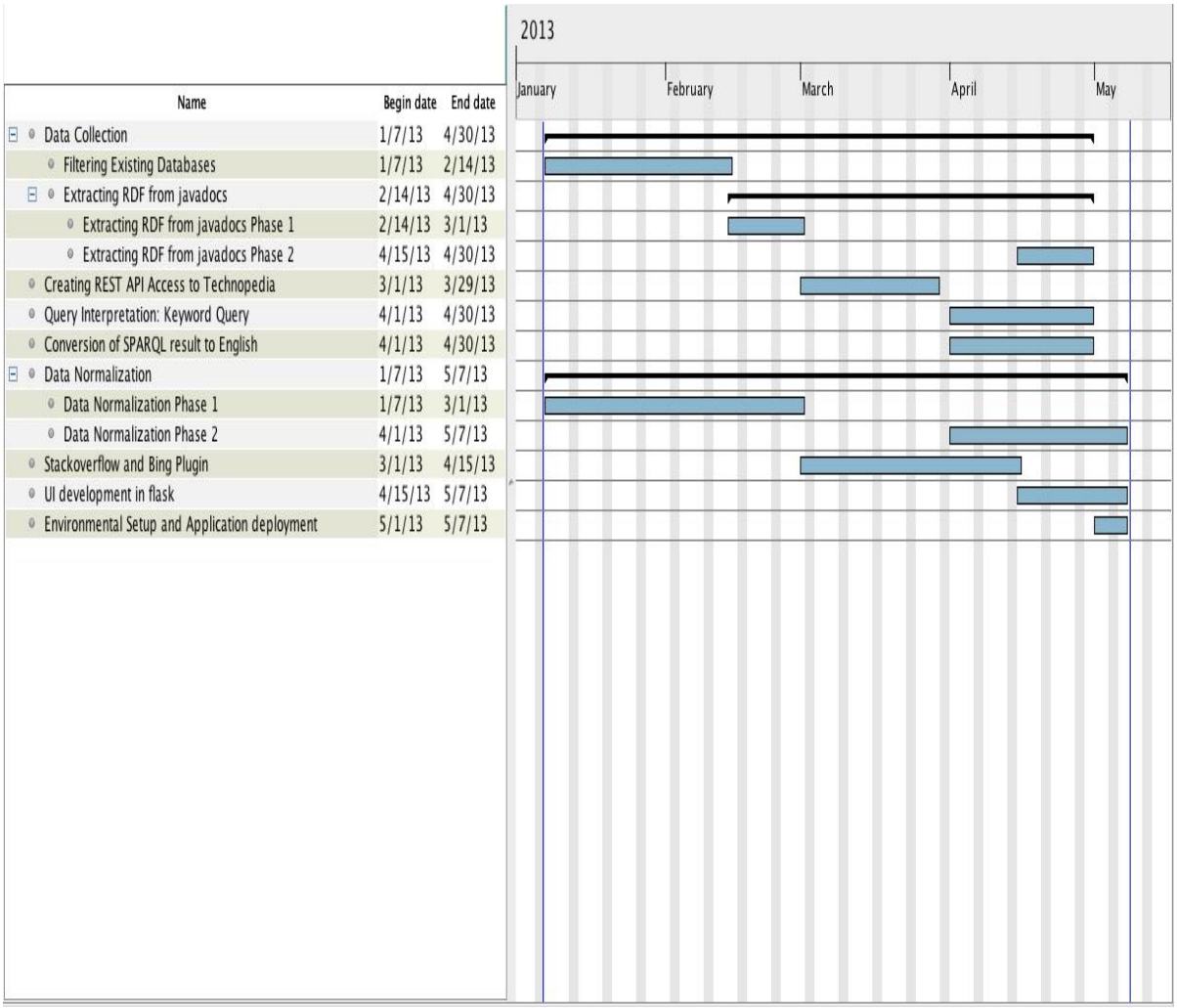
10.3 Test Report Details

Function ality	TC Description	Steps	Expected Results	Actual Results	Result Pass /Fail	Error details	Comments and status
Dataset	Completeness of Javadocs	Check if all the packages, classes, interfacing, annotations, errors and exceptions are present.	All quads should be present in the dataset.	The dataset is complete.	PASS	None	Templates are not present because <<>> pose a problem. Blacklisted for future incorporation into the dataset.
Dataset	Freebase and DBPedia	All entries related to Computers and related fields are to be present after filtering. The dataset is checked randomly for inclusion of topics that ought to be in the dataset.	>90% of the relevant data should be retained after filtering.	The dataset is complete and has retained all required quads.	PASS	None	NA
REST API	Results format and correctness	The results returned by the Technopedia REST API should be in JSON format.	The JSON object returned should validate on any online validation service.	Correctly validated	PASS	None	NA
Query Interpretation	The generated query should be syntactically and semantically correct.	The SPARQL query is validated against any online SPARQL query validation service. Logical correctness is to be checked.	The validation should be successful.	Correctly validated	PASS	None	NA
Kappa	The results are to be displayed as English-like facts.	A search is carried out using the Kappa engine and the results are observed.	The results returned should be accurate and lead to the correct URLs for more information.	The facts are churned out by the engine. They are correct and the URLs are functional.	PASS	Does not work for parameters of functions	NA
Kappa	Bing and Stackoverflow plugins are used to display additional results.	The plugins are integrated properly and the results are returned in few seconds. This should be seen in the UI.	The results should be neatly displayed on the screen in a matter of seconds.	Works as expected	PASS	None	NA

11
Gantt Chart

"We can chart our future clearly and wisely only when we know the path which has led to the present."

- Adlai E. Stevenson



12
Conclusion

"When asked how he knew a piece was finished, he responded, 'When the dinner bell rings'"
— Apocryphal anecdote about Alexander Calder

We carried out a systematic study on the landscape of Linked Open Data[28] and the Semantic Web. We also justified the need for a Linked Open Dataset focused on purely technical data. We studied the RDF standards in [29] and learnt about the intricate and complex structure of the Semantic Layer Cake[30]. These exercises broadened our understanding of the scope and goals of the project.

Since our primary aim was to create a dataset, that was our starting point. After filtering existing datasets, we created a Crawler and used Screenscraping to extract RDF quads from semi-structured documentation. We normalized the dataset to ensure its compatibility with existing datasets.

A lot of analysis was carried out in order to study how queries can be interpreted. Numerous papers were studied and we created our own hybrid method incorporating the best techniques from them. During the process of implementation we discovered a wealth of Python packages that lent themselves to our purpose. We also learnt about how SPARQL can be effectively used to query RDF data.

Our last objective was to create an application that would serve as a proof-of-concept. We created a user friendly web browser interface. In the backend, our dataset was designed to serve as a backbone from which relevant facts were generated as query results. We also incorporated plugins from Bing and StackOverflow to achieve a higher intuitive degree of recall. A mind map was added to facilitate visualization of knowledge from the Technopedia dataset.

The length, breadth and depth of our domain of studyis vast; it's potential, immeasurable. A lot of techniques were tried and tested, but many more techniques are yet to be uncovered. As far as this project is concerned, the dinner bell rang when a reasonably large structured technical dataset was obtained and whose potential was been illustrated by a simple yet meaningful and useful application, a fact finding engine.

13

Future Enhancements

"The best way to predict the future is to invent it."

– Allan Kay

Semanticizing technical documents is a very new idea. There is a lot of scope of exploration and enhancement. The methodology of semanticizing the web is one of the hot research topics to be explored. This could bring upon a revolution in the way web search, IR and machine learning works.

In the following section, we present a few ideas for future enhancement:

- There is no LOD of a technical documentation available till date. The dataset can be converted into an LOD adhering to the standards. It is already made available to the public through a REST API.
- The dataset can be further enlarged and enriched to include a wider set of programming languages. Crowdsourcing can be a good approach.
- The data set can be made available for all other popular languages. We already have Javadocs ready.
- A set of predicates were added specifically to support the semanticizing of technical documentation. These predicates can be standardized to bring in global acceptance.
- When a data set is big, the system resources available are not enough to support the data efficiently. A remote server with higher capabilities will speed up the query fetching. A distributed system holding the data could also help in enhancing the performance.
- Query interpretation algorithm works great; it can be further enhanced by converting the semi-English results into to proper English.
- Optimization techniques can be applied to the commonly asked queries by the users. This will improve the performance of the fact engine.

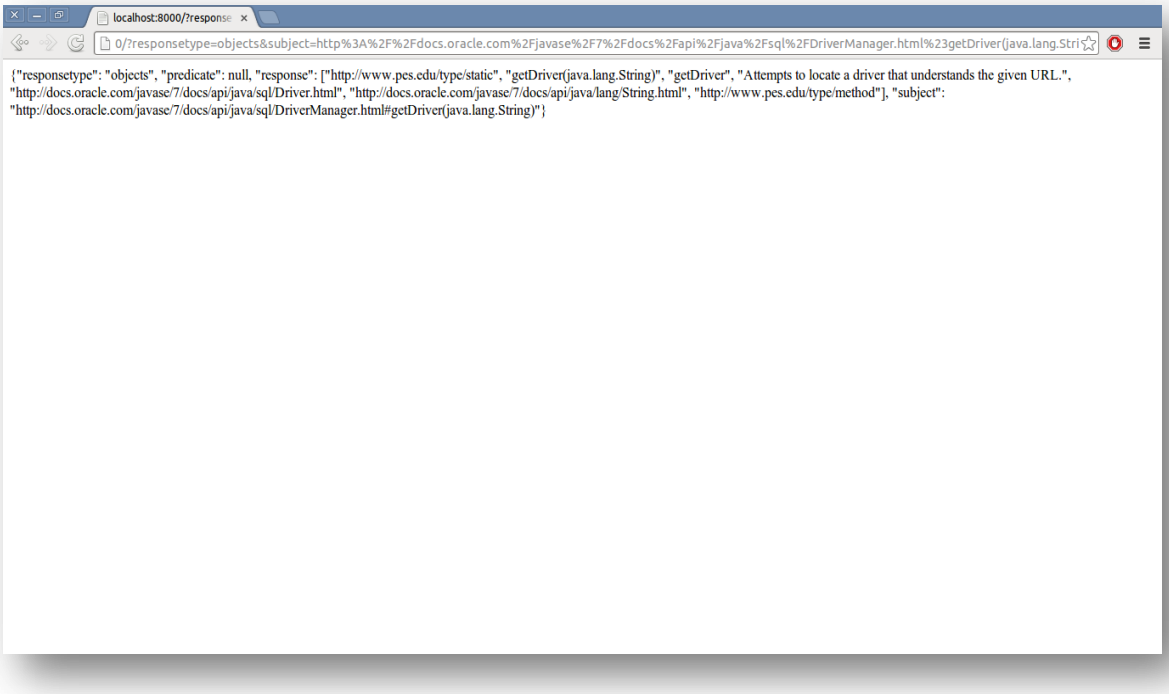
The above cover a few enhancements with respect to the application: Kappa. A lot more interesting, innovative and useful applications can be built using the Technopedia dataset. SWITH serves as a proof of concept that semanticizing technical documentation can help IT professionals.

14

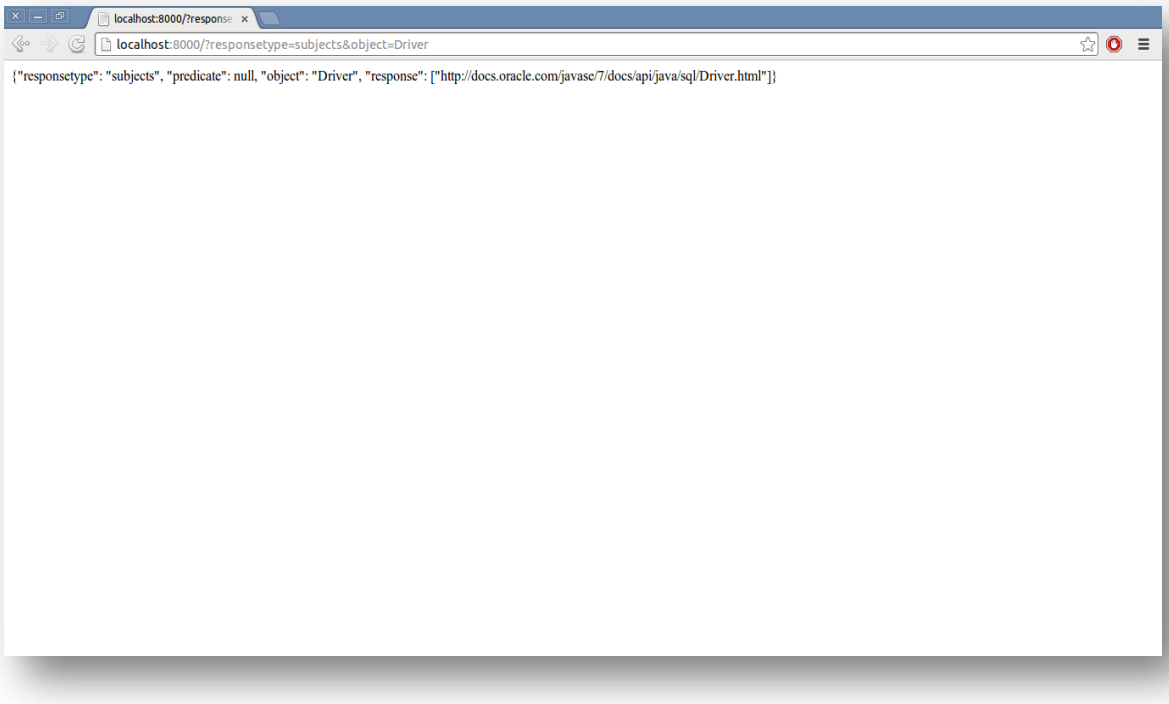
Screenshots

"I have yet to meet a man as fond of high moral conduct as he is of outward appearances."

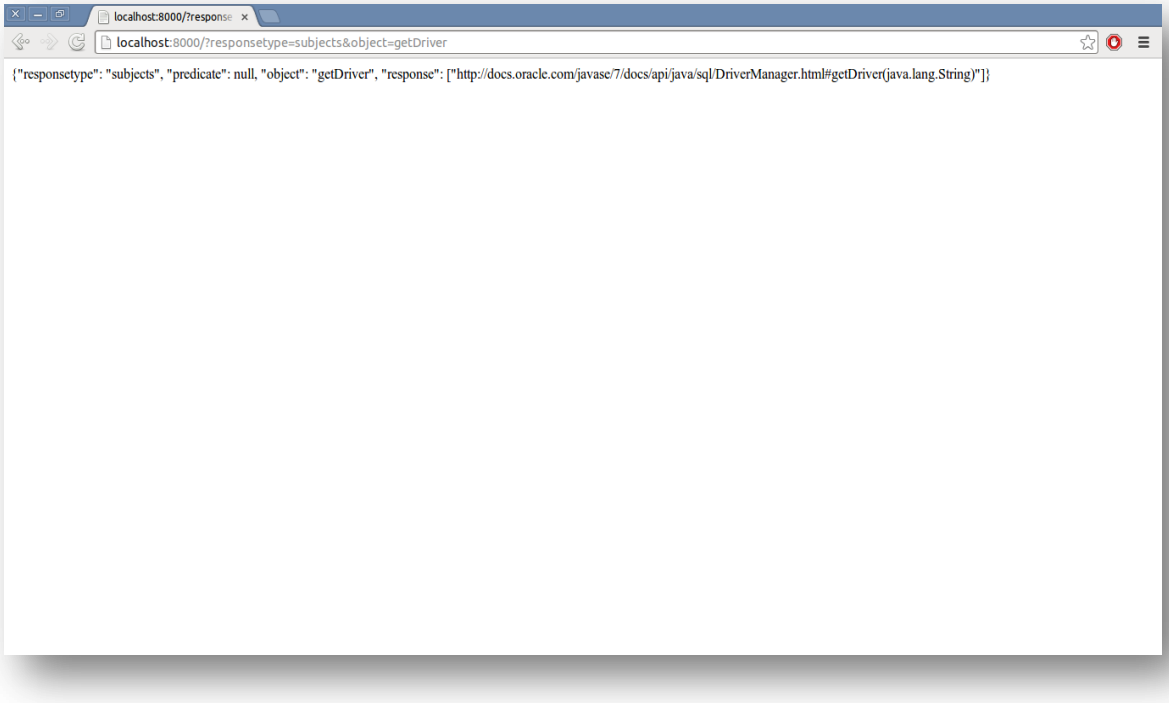
– Confucius



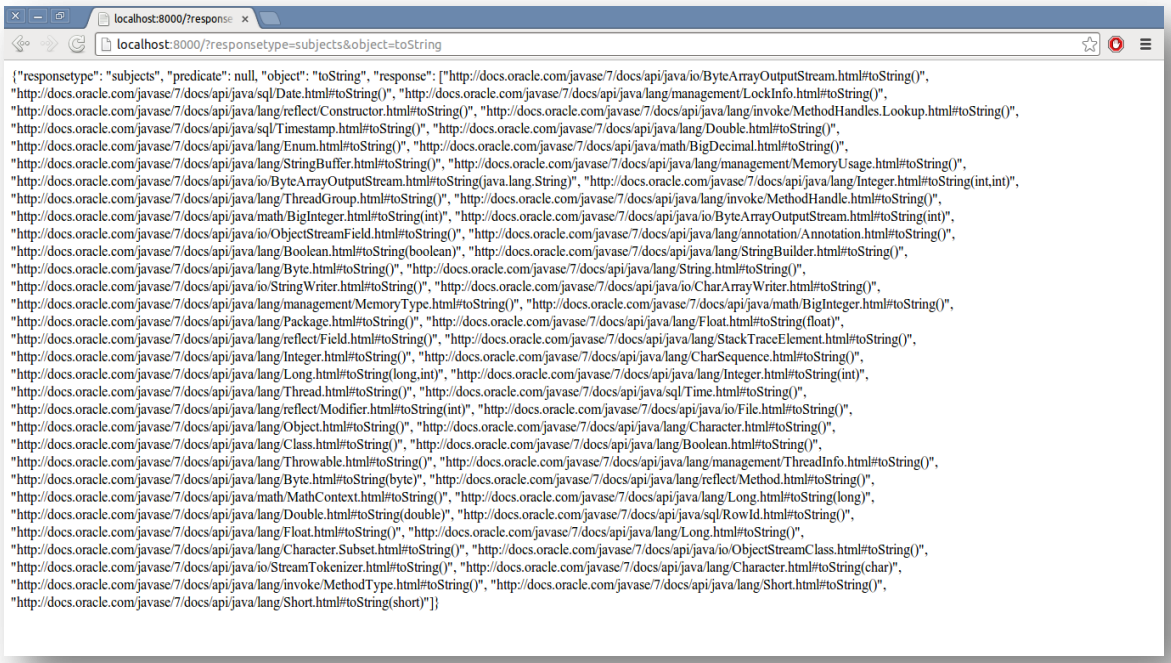
Screenshot 1REST API- Object given Subject



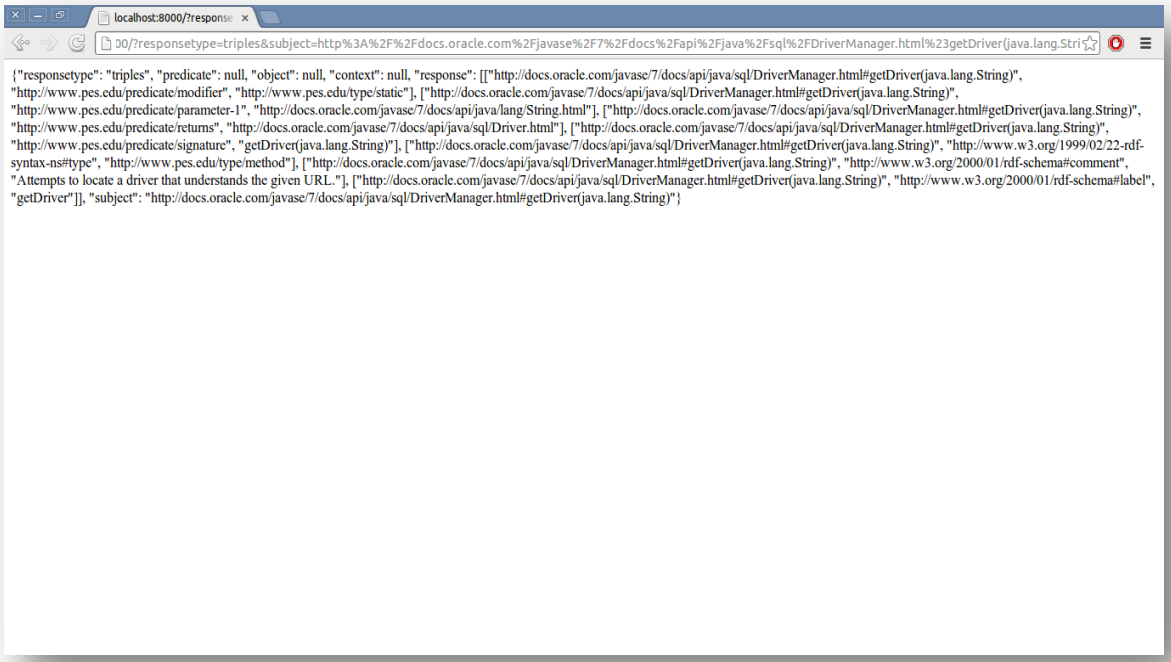
Screenshot 2 REST API- Subject given Object 1



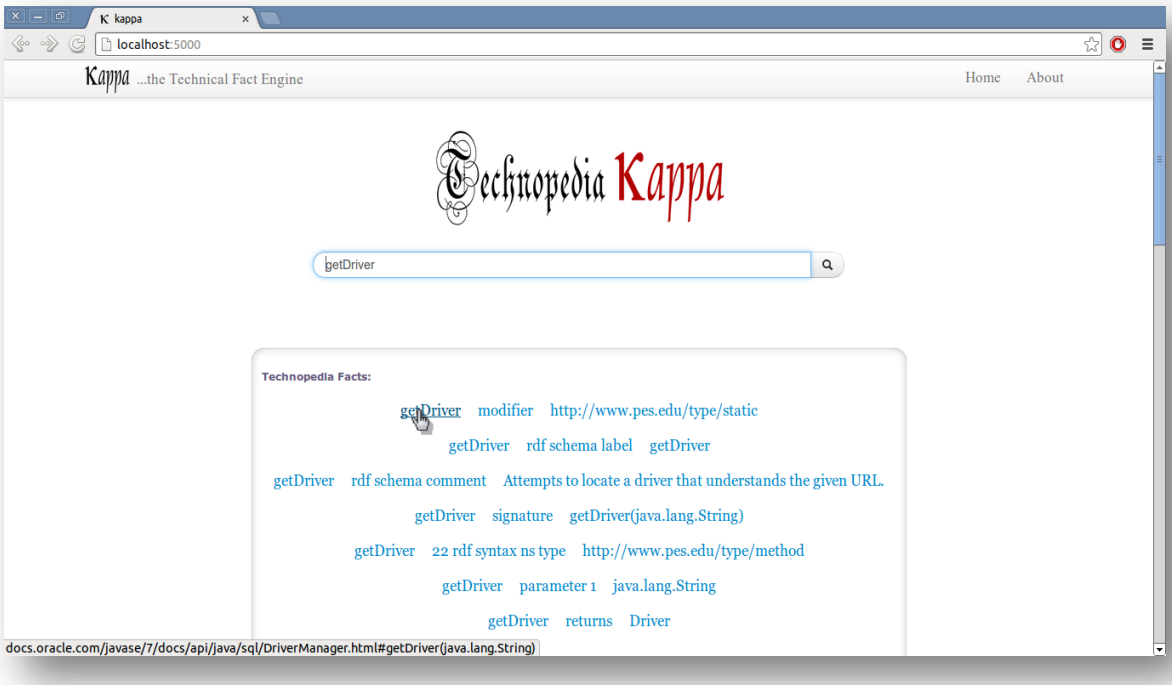
Screenshot 3 REST API- Subject given Object 2



Screenshot 4 REST API- Subject given Object 3

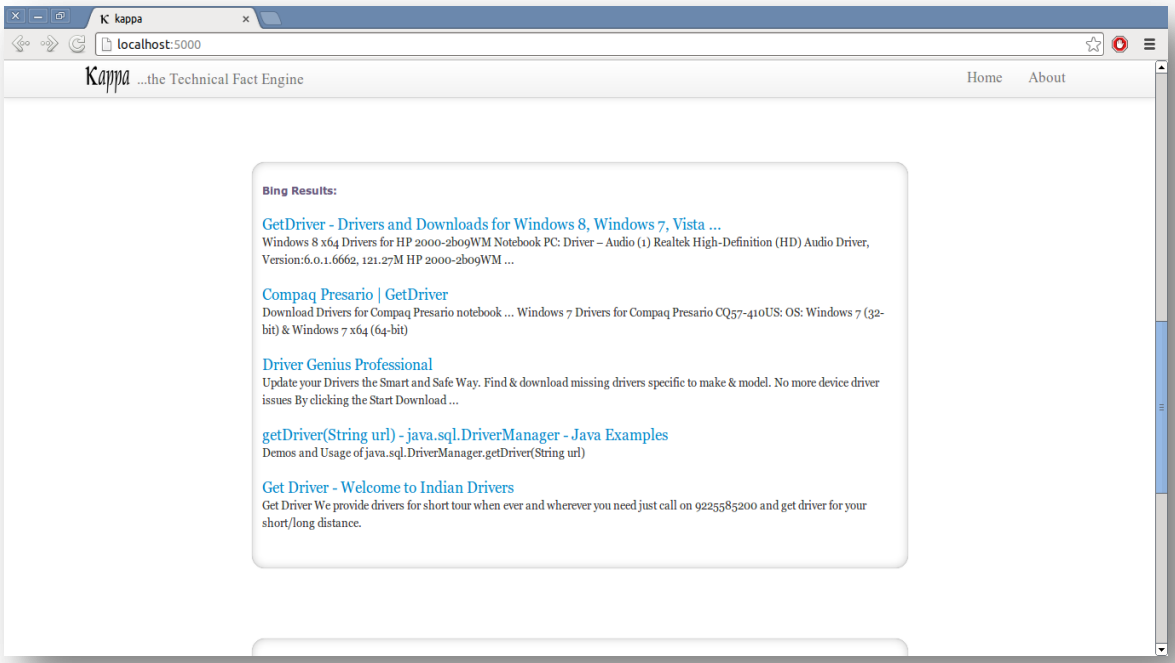


Screenshot 5 REST API- Triple given Subject

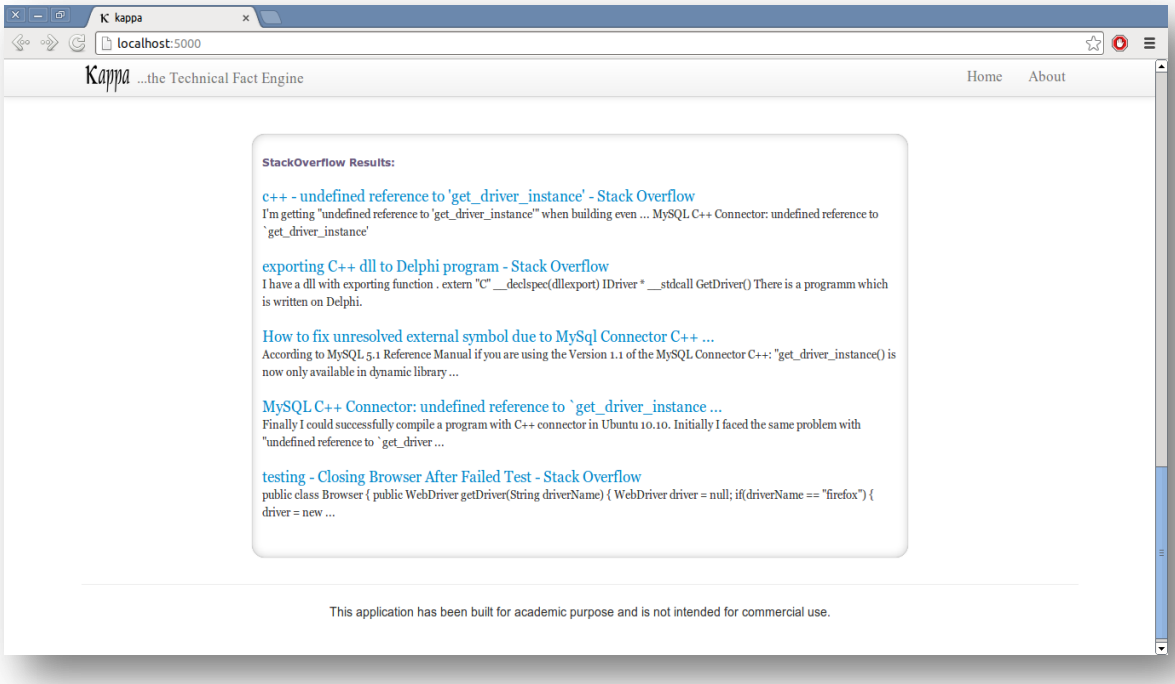


Screenshot 6 User Interface- Technopedia Facts

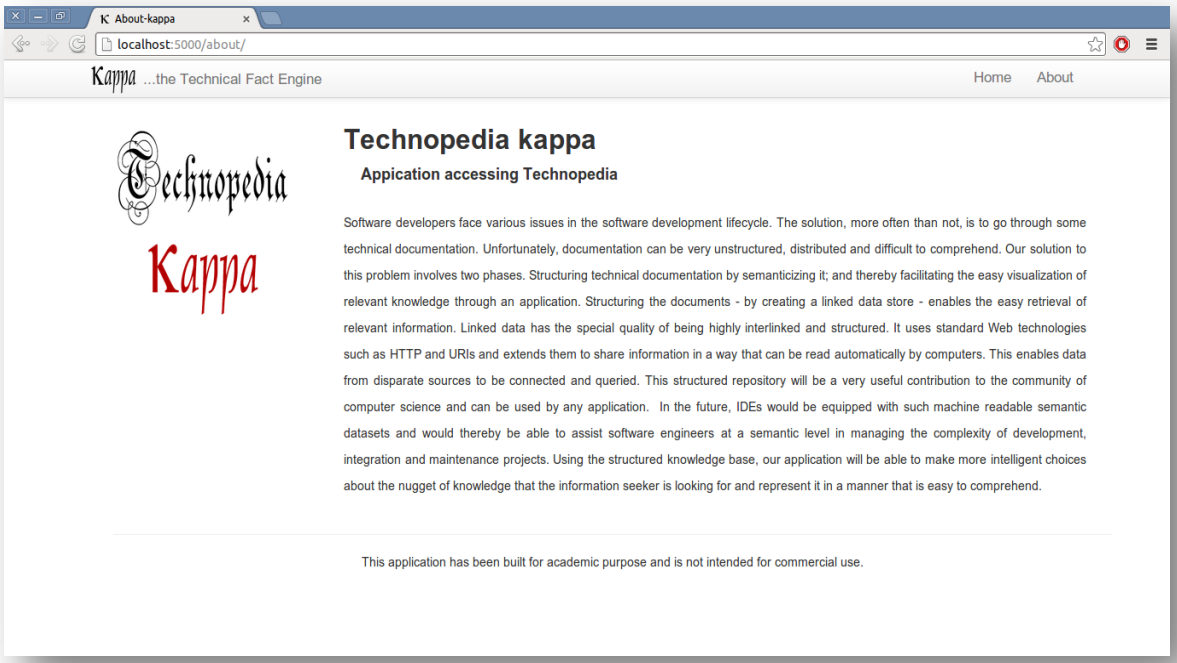
Screenshot 7 User Interface- Facts and Mindmap



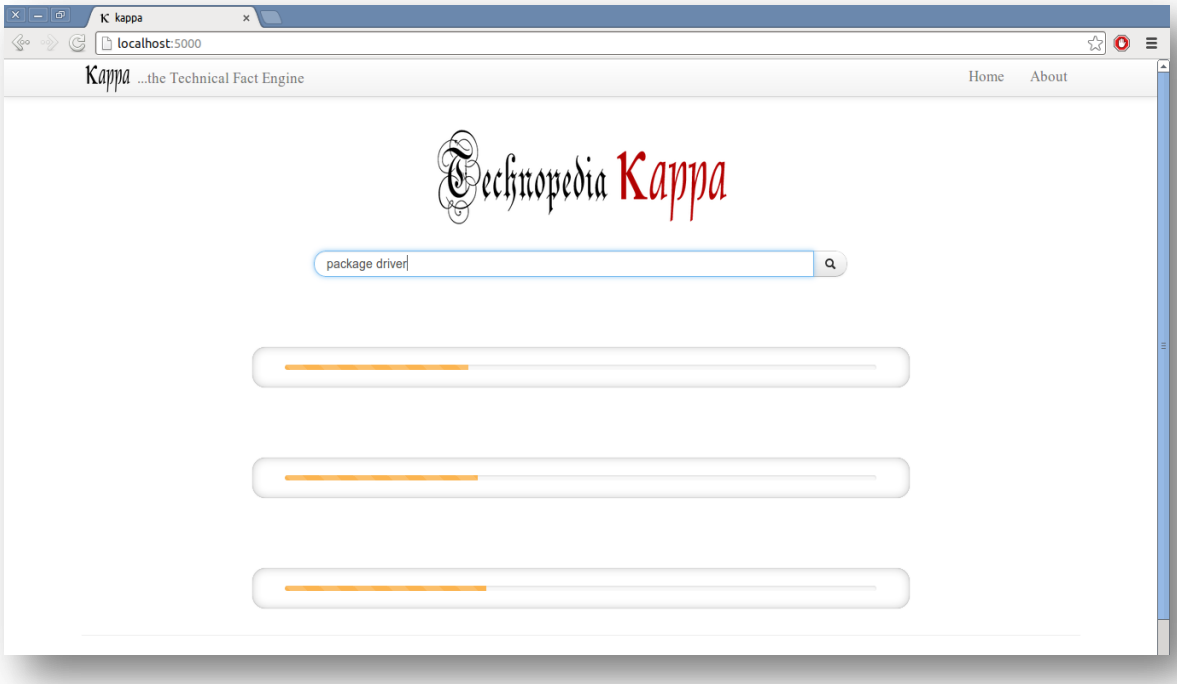
Screenshot 8 User Interface- Bing Results



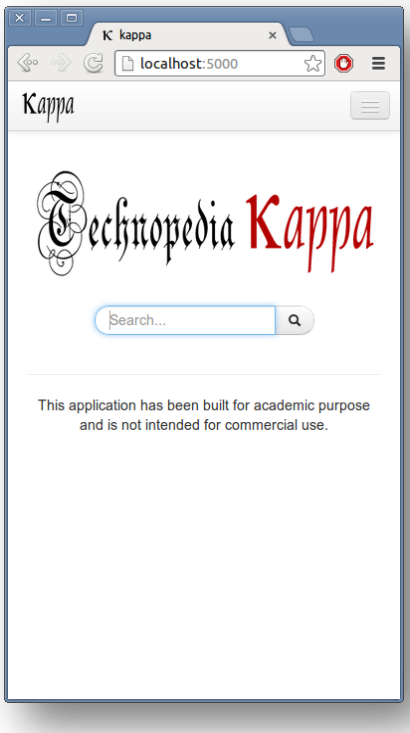
Screenshot 9 User Interface- StackOverflow Results



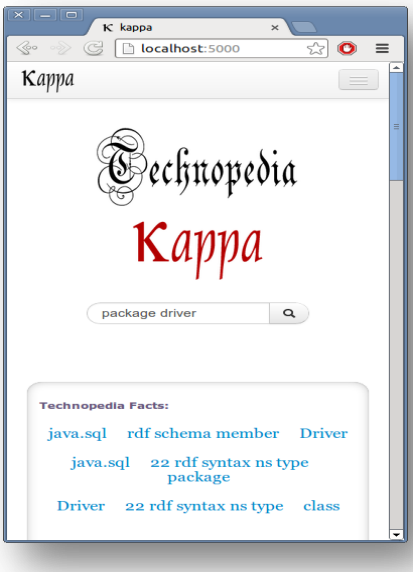
Screenshot 10About Page



Screenshot 11 User Interface- Loading...



Screenshot 12 Mobile Interface – Search



Screenshot 13 Mobile Interface- Results

BIBLIOGRAPHY

[1] "***Semantic University - Cambridge Semantics***." *Semantic University - Cambridge Semantics*.N.p., n.d. Web. 02 May 2013.

[2] Fakhraee, Sina, and FarshadFotouhi. "***Effective Keyword Search over Relational Databases Considering keywords proximity and keywords N-grams***."*Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*. IEEE, 2011

[3] Haam, Deokmin, Ki Yong Lee, and MyoungHo Kim. "***Keyword search on relational databases using keyword query interpretation***." *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*.IEEE, 2010.

[4]F. Siasar Djahantighi, M. Norouzifard, SHDavarpanah, MH Shenassa (2008) "***Using Nat ural language processing in order to create SQL queries***", Proceedings of the International Conference on Computer and Communication Engineering, Kuala Lumpur, Malaysia.

[5] Adolphs, Peter, et al. "***Yago-qa: Answering questions by structured knowledge queries***." *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*. IEEE, 2011.

[6] Shekarpour, Saeedeh, et al. "***Keyword-driven sparql query generation leveraging background knowledge***." *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*.Vol. 1.IEEE, 2011.

[7] Tran, Thanh, et al. "***Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data***." *Data Engineering, 2009.ICDE'09.IEEE 25th International Conference on*.IEEE, 2009.

[8] MySQL, A. B. "***MySQL database server***." *Internet WWW page, at URL: http://www. mysql. com (last accessed/1/00)* (2004).

[9] Krech, Daniel. "***RDFLib***." (2010).

[10] Richardson, Leonard. "***Beautiful Soup Documentation***." (2007).

[11] Richardson, L. "***Beautiful Soup-HTML***." *XML parser for Python* (2008).

[12] Loper, Edward, and Steven Bird. "***NLTK: The natural language toolkit***."*Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*.Association for Computational Linguistics, 2002.

[13] Nurseitov, Nurzhan, et al. "***Comparison of JSON and XML data interchange formats: A case study.***" *Computer Applications in Industry and Engineering (CAINE)* (2009): 157-162.

[14] Hagberg, A., D. Schult, and P. Swart. "***Networkx: Python software for the analysis of networks".***Technical report, Mathematical Modeling and Analysis, Los Alamos National Laboratory, 2005. http://networkx. lanl. gov, 2005.

[15] Hunter, John D. "***Matplotlib: A 2D graphics environment***." *Computing in Science & Engineering* (2007): 90-95.

[16] Duerst, M., and M. Suignard. "***RFC 3987***." *Internationalized resource identifiers (IRis)* 1 (2005).

[17] Loper, Edward. "***Epydoc: API documentation extraction in Python***." *URL: http://epydoc. sourceforge. net/pycon-epydoc. ps. Accessed* 13 (2008).

[18] Prud'Hommeaux, Eric, and Andy Seaborne. "***SPARQL query language for RDF***." *W3C recommendation* 15 (2008).

[19] "***Wiki.dbpedia.org : About***." *Wiki.dbpedia.org : About*. N.p., n.d. Web. 02 May 2013. <http://dbpedia.org/>.

[20] "***A Community-curated Database of Well-known People, Places, and Things***." *Freebase*. N.p., n.d. Web. 02 May 2013. <http://www.freebase.com/>.

[21] "***Java™ Platform, Standard Edition 7 API Specification***." *Overview (Java Platform SE 7)*.N.p., n.d. Web. 02 May 2013. <http://docs.oracle.com/javase/7/docs/api/overview-summary.html>.

[22] ***Jena, O. W. L.*** "***API***." *URL: http://jena. apache. org/tutorials/rdf_a pi. html*.

[23] Broekstra, Jeen, ArjohnKampman, and Frank Van Harmelen. "***Sesame: A generic architecture for storing and querying rdf and rdf schema.***" *The Semantic Web—ISWC 2002*.Springer Berlin Heidelberg, 2002.54-68.

[24] Blakeley, C. "***Virtuoso rdf views-getting started guide***." *OpenLinkSoftware*(2007).

[25] Steve Harris "***4store - Scalable RDF storage***." (2009)

[26] Newman, Chris. ***SQLite (Developer's Library)***.Sams, 2004.

[27] "***Difflib — Helpers for Computing Deltas***." *7.4. Difflib*. N.p., n.d. Web. 06 May 2013. <http://docs.python.org/2/library/difflib.html>.

[28] Cyganiak, Richard, and AnjaJentzsch. "***Linking open data cloud diagram***." *LOD Community (http://lod-cloud. net/)* (2011).

[29] "***RDF Tutorial***." *RDF Tutorial*. N.p., n.d. Web. 02 May 2013. <http://www.w3schools.com/rdf/>.

[30]Nowack, Benjamin. "***The Semantic Web - Not a Piece of Cake.***" *Benjamin Nowack's Blog*.N.p., 8 July 2009. Web. 02 May 2013. <http://www.bnode.org/blog/2009/07/08/the-semantic-web-not-a-piece-of-cake>.

APPENDIX A

API usage guide and implementation details

Module: data.py

An interface to access the technopedia dataset which is loaded on a MySQL database. It creates an rdflib graph and store and initialises them. Effectively wrapping itself around the rdflib objects.

Classes	[hide private]
	Term A class which defines static functions to manipulate term information.
	ParseError
	SparqlError

Functions	[hide private]
	__init__(name=_NAME, connection_str=_CONN_STR) Initialises the graph by loading the database onto it.
	_sparql_query(query_string, initNs={}, initBindings={}) Private method to execute a SPARQL query on technopedia.
	_stringify_sparql_result(res, format) Prefix bnode value with _: Example: N69bdff2a33874675b1b02 => _:N69bdff2a33874675b1b02
	query(query_string, format="json") Method to execute a SPARQL query on technopedia.
	subjects(predicate=None, object=None, format="python") List of subjects for given predicate and/or object.
	predicates(subject=None, object=None, format="python") List of predicates for given subject and/or object.
	objects(subject=None, predicate=None, format="python") List of objects for given predicate and/or subject.
	contexts(subject=None, predicate=None, object=None, format="python") List of contexts in which given subject and/or predicate and/or object appear
	triples(subject=None, predicate=None, object=None, context=None, format="python") List of triples in which given subject and/or predicate and/or object and/or context appear.
	literals(subject=None, predicate=None, format="python") List of literals for given predicate and/or subject.

Variables [hide private]

	_NAME = "technopedia_javadocs"
	_CONN_STR = "host=localhost,user=root,password=root"
	_graph = None

Function Details [hide private]

__init__(name=_NAME, connection_str=_CONN_STR) *(Constructor)*

```
Initialises the graph by loading the database onto it.
@param:
    id :: identifier for the Technopedia object
conn_str :: To connect to the database in format
                host=address,user=name,password=pass,db=tech_fb
```

_sparql_query(query_string, initNs={}, initBindings={})

Private method to execute a SPARQL query on technopedia. Calls rdflibgraph.query() internally.

Handles errors and exceptions without exposing rdflib.

Returns:
SPARQL result as rdflib_sparql.processor.SPARQLResult object.

query(query_string, format="json")

Method to execute a SPARQL query on technopedia.

Returns:
SPARQL result in xml or json format as required. Default is json.

subjects(predicate=None, object=None, format="python")

```
List of subjects for given predicate and/or object.

@param:
    predicate :: as string
    object :: as string
If none given, all subjects will be returned

Returns ::
a python dictionary when format="python" (default)
    JSON object when format="json"
    The return value is of the form:
        {
          "responsetype": "subjects",
          "object": "object_value",
          "predicate": "predicate_value",
          "response": [list of subjects in str format]
        }
```

predicates(subject=None, object=None, format="python")

```
List of predicates for given subject and/or object.

@param:
    subject :: as string
    object :: as string
If none given, all predicates will be returned

Returns ::
    a python dictionary when format="python" (default)
    JSON object when format="json"

    The return value is of the form:
        {
          "responsetype": "predicates",
          "subject": "subject_value",
          "object": "object_value",
          "response": [list of predicates in str format]
        }
```

```
objects(subject=None, predicate=None, format="python")

List of objects for given predicate and/or subject.

@param:
    subject :: as string
    predicate :: as string
If none given, all objects will be returned

Returns ::
    a python dictionary when format="python" (default)
    JSON object when format="json"

    The return value is of the form:
        {
          "responsetype": "objects",
          "subject": "subject_value",
          "predicate": "predicate_value",
          "response": [list of objects in str format]
        }
```

```
contexts(subject=None, predicate=None, object=None,
format="python")

List of contexts in which given subject and/or predicate and/or object appear

@param:
    subject :: as string
    predicate :: as string
    object :: as string
If none given, all contexts will be returned

Returns ::
    a python dictionary when format="python" (default)
    JSON object when format="json"
    The return value is of the form:
        {
          "responsetype": "contexts",
          "subject": "subject_value",
```

```
          "object": "object_value",
          "predicate": "predicate_value",
          "response": [list of contexts in str format]
        }
```

triples(subject=None, predicate=None, object=None, context=None, format="python")

```
List of triples in which given subject and/or predicate and/or        object
and/or context appear.

@param:
    subject :: as string
    predicate :: as string
    object :: as string
    context :: as string
If none given, all triples will be returned

Returns ::
    a python dictionary when format="python" (default)
    JSON object when format="json"

    The return value is of the form:
        {
          "responsetype": "triples",
          "subject": "subject_value",
          "object": "object_value",
          "predicate": "predicate_value",
          "context": "context_value",
          "response": [list of triples] each triple is a list of string
        }
```

literals(subject=None, predicate=None, format="python")

```
List of literals for given predicate and/or subject.

@param:
    subject :: as string
    predicate :: as string
If none given, all literals will be returned

Returns ::
    a python dictionary when format="python" (default)
    JSON object when format="json"

    The return value is of the form:
        {
          "responsetype": "literals",
          "subject": "subject_value",
          "predicate": "predicate_value",
          "response": [list of literals in str format]
        }
```

APPENDIX B

API usage guide and implementation details

Module:kappa.py

An application built on top of Technopedia API to generate facts given a query by using an explorative graph search algorithm.

Classes	[hide private]
	_GE a node is of form ("node", key) an edge is of form ("edge", node1, node2, key)
	_Cursor

Functions	[hide private]
	_get_keyword_index() function which maps keyword to Graph Elements (_GE objects) the graph elements are cnodes, vnodes, redges and aedges
	_extract_keywords_from_literal(literal) function that fetches keywords from literal
	_extract_keywords_from_uri(uri) function that fetches keywords from URI
	_clean_hash(keyword) function that looks for hashes and seperates the keywords
	_clean_camelCase(keyword) function looks for camelCase in the keywords and uncamelCases them
	_get_summary_graph()
	_attach_costs(graph) function which attaches the cost to every node and edge of the graph Returns : graph with costs attached
	_attach_node_costs(graph) function which attaches the cost to every node of the graph...
	_get_node_cost(node, graph, total_number_of_nodes) function which returns the cost associated with the node popularity score: section 5 @param: node: node in the summary graph graph: a graph to which the edge belongs to total_number_of_nodes: total_number_of_nodes in the graph Returns :: a score in the range 0-1
	_attach_edge_costs(graph)

	function which attaches the cost to every edge of the graph...
	_get_edge_cost(edge, graph, total_number_of_edges) function which returns the cost associated with the node popularity score :section 5
	_get_subgraph_cost(graph) function which returns the cost of the subgraph
	_preprocess(query) function pre processes the query removing stop words and punctuation
	_get_in_built_match_score(s) function computes the score of how well the n-gram matches the given key word using a built in scoring technique
	_get_custom_match_score(s) function computes the score of how well the n-gram matches the given key word using a custom function
	_query_to_keywords(query)
	_query_to_keyword(query) function matches the keyword query string to the keyword index
	_get_keyword_elements(keyword_list) function returns a list of list of keyword elements given a list of keywords.
	_make_augmented_graph(K) function which makes an augmented summary graph.
	_alg1(num, dmax, K)
	_remove_ge(ge, ge_list) function to remove a GraphElement (_GE obj) from a list of _GE objects based on the key.
	_alg2(n, m, LG, LQ, k, R)
	_is_connected(n, m) checks if the graph element is connected to all the keywords along the path already visited which is tracked by the cursors.
	_cursor_combinations(n) function to obtain a combination of paths to graph element - n, originating from different keywords.
	_build_m_paths(m_cursor_list) A function that builds a set of paths for the given list of m cursor list
	_build_path_from_cursor(cursor) function to obtain a path from a cursor
	_merge_paths_to_graph(paths)

	function to merge paths to a subgraph
	_update_cost_of_subgraph(subgraph) function to update the cost of subgraph
	_choose_top_k_sub_graphs(cost_augmented_subgraph_list, k) function to fetch the best k subgraphs
	_ret_cost(a) function to return
	_k_ranked(LG, k) function to return the kth ranked element
	_map_to_query(G) function which maps a graph to a conjunctive query.
	_initialize_const_var(subgraph)
	_map_edge(e, node_dict) function which maps a given edge to a set of conjunctive query predicates @param e :: a networkx edge with key info node_dict :: a dictionary of nodes and their associated variables
	_get_sparql_query(conj_query) function which returns a sparql query given a conjuunctive query
	_parse_query(query_string) Returns: lol_sparql_query::A list of list that contains the where part query split on "."
	_parse_query1(query_string) function which cleans and parses the sparql query
	_filter_predicates(lol_sparql_query) function that fetches the relevant triples from the SPARQL query
	_extract_facts(var_dict, bindingsList, lol_filtered, j) function to extract facts from a given list of bindings, variables and required predicates
	topk(query, num) function which accepts a keyword query from the user, processes it and runs the topk algorithms on it.
	sparql_to_facts(sparql_query) Function to parse the SPARQL query and the JSON result
	conj_to_facts(conj_query) Function to parse the CONJUNCTIVE query and the JSON result
Variables	[hide private]
	_connecting_ele = 0
	_keyword_index = _get_keyword_index()

	`_summary_graph = _attach_costs(_summary_graph)`

Function Details

[hide private]

_get_keyword_index()

```
function which maps keyword to Graph Elements (_GE objects)
the graph elements are cnodes, vnodes, redges and aedges

Returns :
index dictionary of form {"keyword":[list of _GE objects],...}
```

_extract_keywords_from_literal(literal)

function that fetches keywords from literal

Returns:
returns a list of keywords keyword_list = [k1,k2,k3,..kn]

_extract_keywords_from_uri(uri)

function that fetches keywords from URI

Returns:
returns a list of keywords keyword_list = [k1,k2,k3,..kn]

_clean_hash(keyword)

function that looks for hashes and seperates the keywords

Returns:
pre_hash_keyword:: the key word before "#"
post_hash_keyword::the key word after "#"

_clean_camelCase(keyword)

function looks for camelCase in the keywords and uncamelCases them

Returns:
The cleaned keyword

_attach_node_costs(graph)

```
function which attaches the cost to every node of the graph
@param
    graph :: networkx graph without cost info
Returns :
networkx graph with node costs attached
```

_get_node_cost(node, graph, total_number_of_nodes)

```
function which returns the cost associated with the node
popularity score: section 5
@param:
    node: node in the summary graph
    graph: a graph to which the edge belongs to
total_number_of_nodes: total_number_of_nodes in the graph
Returns ::
    a score in the range 0-1
```

```
cost of a BNode and Thing is 0
```

_attach_edge_costs(graph)

```
function which attaches the cost to every edge of the graph
@param
    graph :: networkx graph without cost info
Returns :
    graph with edge costs attached
```

_get_edge_cost(edge, graph, total_number_of_edges)

function which returns the cost associated with the node popularity score :section 5

Returns:
a score in the range 0-1

_get_subgraph_cost(graph)

function which returns the cost of the subgraph

Returns:
a cumilative score

_preprocess(query)

function pre processes the query removing stop words and punctuation

Returns:
The cleaned keyword query

_get_in_built_match_score(s)

function computes the score of how well the n-gram matches the given key word using a built in scoring technique

Returns:
score:: a floating value 0.0-1.0 threshold::a threshold value for the score

_get_custom_match_score(s)

function computes the score of how well the n-gram matches the given key word using a custom function

Returns:
score:: a floating value 0.0-1.0 threshold::a threshold value for the score

_query_to_keyword(query)

function matches the keyword query string to the keyword index

Returns:
The dictionary of top ten matching keywords from the index

_get_keyword_elements(keyword_list)

```
function returns a list of list of keyword elements given
a list of keywords.

uses the _keyword_index data structure

@param
keyword_list :: list of keywords(strings)
    [keyword1, keyword2, ...]

Returns :
    K :: list of list of _GE objects (keyword elements)
    [[_GE objects for keyword1], [_GE objects for keyword2], ...]
```

_make_augmented_graph(K)

```
function which makes an augmented summary graph.

takes a copy of the _summary_graph and adds vnodes and aedges to it.
attaches the augmented graph to _GE.graph

@param
    K :: list of list of _GE objects (keyword elements)

Returns :
aug_graph :: networkx graph with keyword elements attached
```

_remove_ge(ge, ge_list)

```
function to remove a GraphElement (_GE obj) from a list of _GE objects
based on the key.

used to remove the parent _GE from the list of neighbours _GEs

@param
ge :: _GE object
g_list :: list of _GE objects
returns nothing; simply alters the list of _GEs
```

_is_connected(n, m)

```
checks if the graph element is connected to all the keywords along the
path already visited which is tracked by the cursors.

n is a connecting element if all n.Ci are not empty, i.e.
for every keyword i, there is at least one cursor

@param
    n :: graph element
    m :: no of keywords as int
Returns :
boolean value
```

_cursor_combinations(n)

function to obtain a combination of paths to graph element - n, originating from different keywords.

each element has a list of list of cursors. each element will have m lists; one list for each keyword. each of these lists may have more than one cursor to the keyword i. each cursor represents a path from element to keyword i.

this function therefore returns a list of tuples where each tuple has a one cursor to each keyword. the length of every tuple is therefore m.

the reurned list represents all possible subgraphs from node to each of the keywords along all combinations of paths

Returns:
list of tuples of m-cusor paths
[(c11,c12,..c1m),(c21,c22,..c2m),...(ck1,ck2,..ckm)]

_build_m_paths(m_cursor_list)

A function that builds a set of paths for the given list of m cursor list

Returns:
A list of list of m-graph paths

_build_path_from_cursor(cursor)

function to obtain a path from a cursor

Returns:
A path from keyword element to the end node, it is of type mulitDiGraph

_merge_paths_to_graph(paths)

function to merge paths to a subgraph

Returns:
A merged multiDiGraph

_update_cost_of_subgraph(subgraph)

function to update the cost of subgraph

Returns:
A tuple (subgraph,cost)

_choose_top_k_sub_graphs(cost_augmented_subgraph_list, k)

function to fetch the best k subgraphs

Returns:
A list k subgraphs augmented with cost
[(subgraph1,cost1),...(subgraphk,costk)]

_ret_cost(a)

function to return

Returns:
A key, i.e cost

_k_ranked(LG, k)

function to return the kth ranked element

Returns:
Cost of the kth subgraph

_map_to_query(G)

```
function which maps a graph to a conjunctive query.
@param
    G :: networkx graph
Returns ::
conj_query :: a list of conj query predicates
```

_map_edge(e, node_dict)

```
function which maps a given edge to a set of conjunctive query predicates
@param
    e :: a networkx edge with key info
node_dict :: a dictionary of nodes and their associated variables

Returns :
edge_query :: list of conjunctive query predicates for the edge
```

_get_sparql_query(conj_query)

```
function which returns a sparql query given a conjuunctive query

@param
conj_query :: conjunctive query as string
Returns :
sparql_query :: sparql query as string
```

_parse_query(query_string)

Returns:
lol_sparql_query::A list of list that contains the where part query split on "."

_parse_query1(query_string)

function which cleans and parses the sparql query

Returns:
lol_sparql_query::A list of list that contains the where part query split on "."

_filter_predicates(lol_sparql_query)

function that fetches the relevant triples from the SPARQL query

Returns:
:

lol_filtered::A list of list of filtered queries

_extract_facts(var_dict, bindingsList, lol_filtered, j)

function to extract facts from a given list of bindings, variables and required predicates

Returns:
factList:: A dict of facts, each fact is a 3 element list of a dictseg :
[[{subject_lable,subject_uri},{predicate_lable,predicate_uri},{object_lable,object_uri}],[fact2],[fact3],...]

topk(query, num)

function which accepts a keyword query from the user, processes it and runs the topk algorithms on it.

@param
query :: string query from the user
num :: number of interpreattions required (the k in topk)

Returns :
sparql_R :: list of sparql queries

sparql_to_facts(sparql_query)

Function to parse the SPARQL query and the JSON result

Fyzz and yapps parser are used to extract required segments of the SPARQL query given as input

Returns:
factList:: A list of facts, each fact is a 3 element list of a tuple eg :
[[(subject_lable,subject_uri),(predicate_lable,predicate_uri),(object_lable,object_uri)],[fact2],[fact3],...]

conj_to_facts(conj_query)

Function to parse the CONJUNCTIVE query and the JSON result

Returns:
factList:: A list of facts, each fact is a 3 element list of a tuple eg :
[[(subject_lable,subject_uri),(predicate_lable,predicate_uri),(object_lable,object_uri)],[fact2],[fact3],...]